Project Name: Synopse mORMot Framework

Document Name: Software Architecture Design

Document Revision: 1.18

Date: September 16, 2020

Project Manager: Arnaud Bouchez

Document License

Synopse mORMot Framework Documentation.
Copyright (C) 2008-2020 Arnaud Bouchez.
Synopse Informatique - https://synopse.info..

The Synopse mORMot Framework Source Code is licensed under GPL / LGPL / MPL licensing terms, free to be included in any application.

The Synopse mORMot Framework Documentation is a free document, released under a GPL 3.0 License, distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this document uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Prepared by: Arnaud Bouchez
Title: Project Manager

Signature:
Date:

Software Architecture Design - Rev. 1.18
Page 1 of 2552
Document Purpose

The *Software Architecture Design* document purpose is to describe the implications of each software requirement specification on all the affected software modules for the *Synopse mORMot Framework* project.

The current revision of this document is 1.18.

### Related Documents

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Rev.</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWRS</td>
<td>Software Requirements Specifications</td>
<td>1.18</td>
<td>September 16, 2020</td>
</tr>
<tr>
<td>SDD</td>
<td>Software Design Document</td>
<td>1.18</td>
<td>September 16, 2020</td>
</tr>
<tr>
<td>DI</td>
<td>Design Input Product Specifications</td>
<td>1.18</td>
<td>September 16, 2020</td>
</tr>
</tbody>
</table>
# Table of Contents

**Foreword**

- Purpose ............................................. 59
- Responsibilities .................................... 60
  - GNU General Public License ..................... 60

1. Synopse mORMot Overview

- 1.1. Client-Server ORM/SoA framework .................. 74
- 1.2. Highlights ........................................ 75
- 1.3. Benefits .......................................... 76
- 1.4. Legacy code and existing projects .................. 77
- 1.5. FAQ .............................................. 78

2. Architecture principles

- 2.1. General design .................................... 82
- 2.2. Architecture Design Process ....................... 84
- 2.3. Model-View-Controller .............................. 86
- 2.4. Multi-tier architecture ............................. 88
- 2.5. Service-Oriented Architecture (SOA) ............. 90
- 2.6. Object-Relational Mapping (ORM) .................. 92
- 2.7. NoSQL and Object-Document Mapping (ODM) .... 96
- 2.8. Domain-Driven Design ............................. 99
  - 2.8.1. Definition .................................... 99
  - 2.8.2. Patterns ...................................... 99
2.8.3. Is DDD good for you? 100
2.8.4. Introducing DDD 100

3. Enter new territory

3.1. Meet the mORMot 101
3.2. Main units 103

4. SynCommons unit

4.1. Unicode and UTF-8 105
4.2. Currency handling 106
4.3. TDynArray dynamic array wrapper 107
  4.3.1. TList-like properties 107
  4.3.2. Enhanced features 108
  4.3.3. Capacity handling via an external Count 109
  4.3.4. JSON serialization 109
  4.3.5. Daily use 109
  4.3.6. TDynArrayHashed 111
  4.3.7. TSynDictionary 111
4.4. TDocVariant custom variant type 112
  4.4.1. TDocVariant documents 112
    4.4.1.1. Variant object documents 113
    4.4.1.2. FPC restrictions 115
    4.4.1.3. Variant array documents 116
    4.4.1.4. Create variant object or array documents from JSON 117
    4.4.1.5. Per-value or per-reference 117
  4.4.2. Advanced TDocVariant process 118
    4.4.2.1. Number values options 119
    4.4.2.2. Object or array document creation options 119
    4.4.2.3. Integration with other mORMot units 120
4.5. Cross-cutting functions 120
  4.5.1. Iso8601 time and date 120
    4.5.1.1. TDateTime and TDateTimeMS 120
4.5.1.2. TTimeLog and TTimeLogBits
4.5.1.3. TUnixTime and TUnixMSTime

4.5.2. Time Zones

4.5.3. Safe locks for multi-thread applications

4.5.3.1. Protect your resources
4.5.3.2. Fixing TRTCriticalSection
4.5.3.3. Introducing TSynLocker
4.5.3.4. Inheriting from T*Locked
4.5.3.5. Injecting TAutoLocker instances
4.5.3.6. Injecting IAutoLocker instances
4.5.3.7. Safe locked storage in TSynLocker
4.5.3.8. Thread-safe TSynDictionary

5. Object-Relational Mapping

5.1. TSQLRecord fields definition

5.1.1. Property Attributes
5.1.2. Text fields
5.1.3. Date and time fields
5.1.4. TSessionUserID field
5.1.5. Enumeration fields
5.1.6. Floating point and Currency fields
5.1.7. TSQLRecord fields
5.1.8. TID fields
5.1.9. TRecordReference and TRecordReferenceToBeDeleted
5.1.10. TSQLRecord, TID, TRecordReference deletion tracking
5.1.11. Variant fields
5.1.12. Record fields
5.1.13. BLOB fields
5.1.14. TNullable* fields for NULL storage

5.2. Working with Objects

5.3. Queries

5.3.1. Return a list of objects
5.3.2. Query parameters

5.3.3. Introducing TSQLTableJSON

5.3.4. Note about query parameters

5.4. Automatic TSQLRecord memory handling

5.5. Objects relationship: cardinality

5.5.1. "One to one" or "One to many"

5.5.1.1. TSQLRecord published properties are IDs, not instance

5.5.1.2. Transtyping IDs

5.5.1.3. Automatic instantiation and JOINed query

5.5.2. "Has many" and "has many through"

5.5.2.1. Shared nothing architecture (or sharding)

5.5.2.1.1. Embedding all needed data within the record

5.5.2.1.2. Nesting objects and arrays

5.5.2.1.2.1. TDocVariant and variant fields

5.5.2.1.2.1.1. Schemaless storage via a variant

5.5.2.1.2.1.2. JSON operations from SQL code

5.5.2.1.2.2. Dynamic arrays fields

5.5.2.1.2.2.1. Dynamic arrays from Delphi Code

5.5.2.1.2.2.2. Dynamic arrays from SQL code

5.5.2.1.2.3. TPersistent/TCollection fields

5.5.2.1.2.4. Any TObject, including TObjectList

5.5.2.1.2.5. Sharding on NoSQL engines

5.5.2.2. ORM implementation via pivot table

5.5.2.2.1. Introducing TSQLRecordMany

5.5.2.2.2. Automatic JOIN query

5.6. ORM Data Model

5.6.1. Creating an ORM Model

5.6.2. Several Models

5.6.3. Filtering and Validating

5.7. ORM Cache
5.8. Calculated fields

5.8.1. Setter for TSQLRecord

5.8.2. TSQLRecord.ComputeFieldsBeforeWrite

5.9. Audit Trail for change tracking

5.9.1. Enabling audit-trail

5.9.2. A true Time Machine for your objects

5.9.3. Automatic history packing

5.10. Master/slave replication

5.10.1. Enable synchronization

5.10.2. From master to slave

5.10.3. Real-time synchronization

5.10.4. Replication use cases

6. Daily ORM

6.1. ORM is not Database

6.1.1. Objects, not tables

6.1.2. Methods, not SQL

6.1.3. Think multi-tier

6.2. One ORM to rule them all

6.2.1. Rude class definition

6.2.2. Persist TSQLRecord, not any class

6.2.3. Several ORMs at once

6.2.4. The best ORM is the one you need

7. Database layer

7.1. SQLite3-powered, not SQLite3-limited

7.1.1. SQLite3 as core

7.1.2. Extended by SQLite3 virtual tables

7.1.3. Data access benchmark

7.1.3.1. Software and hardware configuration

7.1.3.2. Insertion speed

7.1.3.3. Reading speed
7.1.3.4. Analysis and use case proposal

7.2. SQLite3 implementation

7.2.1. Statically linked or using external dll

7.2.1.1. Static bcc-compiled .obj

7.2.1.2. Official MinGW-compiled sqlite3.dll

7.2.1.3. Visual C++ compiled sqlite3.dll

7.2.2. Prepared statement

7.2.3. R-Tree inclusion

7.2.4. FTS3/FTS4/FTS5

7.2.4.1. Dedicated FTS3/FTS4/FTS5 record type

7.2.4.2. Stemming

7.2.4.3. FTS searches

7.2.4.4. FTS4 index tables without content

7.2.5. Column collations

7.2.6. REGEXP operator

7.2.7. ACID and speed

7.2.7.1. Synchronous writing

7.2.7.2. File locking

7.2.7.3. Performance tuning

7.2.8. Database backup

7.3. Virtual Tables magic

7.3.1. Virtual Table module classes

7.3.2. Defining a Virtual Table module

7.3.3. Using a Virtual Table module

7.3.4. Virtual Table, ORM and TSQLRecord

7.3.5. In-Memory "static" process

7.3.5.1. In-Memory tables

7.3.5.2. In-Memory virtual tables

7.3.5.3. In-Memory and ACID

7.3.6. Redirect to an external TSQLRest

7.3.7. Virtual Tables to access external databases
7.3.8. Virtual tables from the client side

8. External SQL database access

8.1. SynDB direct RDBMS access

8.1.1. Direct access to any RDBMS engine

8.1.2. Data types

8.1.3. Database types

8.1.4. SynDB Units

8.1.5. SynDB Classes

8.1.6. ISQLDBRows interface

8.1.7. Using properly the ISQLDBRows interface

8.1.8. Late-binding

8.1.9. TDataSet and SynDB

8.1.10. TQuery emulation class

8.1.11. Storing connection properties as JSON

8.2. SynDB clients

8.2.1. OleDB or ODBC to rule them all

8.2.2. ZEOS via direct ZDBC

8.2.2.1. The mORMot's best friend

8.2.2.2. Recommended version

8.2.2.3. Connection samples

8.2.3. Oracle via OCI

8.2.3.1. Optimized client library

8.2.3.2. Direct connection without Client installation

8.2.3.3. Oracle Wallet support

8.2.4. SQLite3

8.2.5. DB.pas libraries

8.2.5.1. NexusDB access

8.2.5.2. FireDAC / AnyDAC library

8.2.5.3. UniDAC library

8.2.5.4. BDE engine

8.2.6. Remote access via HTTP
8.2.6.1. Server and Client classes
8.2.6.2. Publish a SynDB connection over HTTP
8.2.6.3. SynDB client access via HTTP
8.2.6.4. Advanced use cases
8.2.6.5. Integration with SynDBExplorer
8.2.6.6. Do not forget the mORMot!

8.3. SynDB ORM Integration

8.3.1. Code-first or database-first
8.3.2. Code-first ORM
8.3.3. Database-first ORM
8.3.4. Sharing the database with legacy code
8.3.5. Auto-mapping of SQL conflictual field names
8.3.6. External database ORM internals
8.3.7. Tuning the process

9. External NoSQL database access

9.1. SynMongoDB client

9.1.1. Connecting to a server
9.1.2. Adding some documents to the collection
9.1.3. Retrieving the documents
9.1.3.1. Updating or deleting documents
9.1.4. Write Concern and Performance

9.2. MongoDB + ORM = ODM

9.2.1. Define the TSQLRecord class
9.2.2. Register the TSQLRecord class
9.2.3. ORM/ODM CRUD methods
9.2.4. ODM complex queries
9.2.5. BATCH mode
9.2.6. ORM/ODM performance

10. JSON RESTful Client-Server

10.1. JSON
10.1.1. Why use JSON? 295
10.1.2. Values serialization 297
10.1.3. Record serialization 297
  10.1.3.1. Automatic serialization via Enhanced RTTI 297
  10.1.3.2. Serialization for older Delphi versions 298
    10.1.3.2.1. Default Binary/Base64 serialization 298
    10.1.3.2.2. Custom serialization 298
    10.1.3.2.3. Defining callbacks 299
    10.1.3.2.4. Text-based definition 300
10.1.4. Dynamic array serialization 303
  10.1.4.1. Standard JSON arrays 303
  10.1.4.2. Customized serialization 304
10.1.5. TSQLRecord TPersistent TStrings TRawUTF8List 306
10.1.6. TObject serialization 306
  10.1.6.1. Custom class serialization 307
  10.1.6.2. Custom field names serialization 308
  10.1.6.3. TObjectList serialization 309
10.2. REST 311
10.2.1. What is REST? 311
  10.2.1.1. Resource-based 311
  10.2.1.2. Unique Identifier 311
  10.2.1.3. Interfaces 312
  10.2.1.4. By Representation 312
  10.2.1.5. Stateless 312
10.2.2. RESTful mORMot 313
  10.2.2.1. BLOB fields 313
  10.2.2.2. JSON representation 314
  10.2.2.3. Stateless ORM 314
10.3. REST and JSON 316
  10.3.1. JSON format density 316
  10.3.2. JSON (not) expanded layouts 316
10.3.3. JSON global cache

11. Client-Server process

11.1. Client-Server cheat sheet

11.2. Protocols

11.3. TSQLRest classes

11.3.1. Server classes

11.3.2. Storage classes

11.3.3. Client classes

11.4. In-process/stand-alone application

11.5. Local access via named pipes or Windows messages

11.6. Network and Internet access via HTTP

11.6.1. HTTP server(s)

11.6.2. High-performance http.sys server

11.6.2.1. Use the http.sys server

11.6.2.2. URI authorization as Administrator

11.6.2.2.1. Secure specific authorization

11.6.2.2.2. Automatic authorization

11.6.2.2.3. Manual URI authorization

11.6.2.3. HTTP API 2.0 Features

11.6.3. HTTP client(s)

11.6.4. HTTPS server

11.6.4.1. Certificates

11.6.4.2. Configure a Port with an SSL certificate

11.6.5. Custom Encodings

11.6.5.1. SynLZ/deflate compression

11.6.5.2. AES encryption over HTTP

11.6.5.3. Prefer WebSockets between mORMot nodes

11.7. Thread-safety

11.7.1. Thread safe design

11.7.2. Advanced threading settings
11.7.3. Proven behavior 337
11.7.4. Highly concurrent clients performance 338

12. Client-Server ORM

12.1. ORM as local or remote 341
12.2. Stateless design 347
   12.2.1. Server side synchronization 347
   12.2.2. Client side synchronization 347
   12.2.3. Let applications be responsive 348
12.3. BATCH sequences for adding/updating/deleting records 350
   12.3.1. BATCH process 350
   12.3.2. Transmitted JSON 352
   12.3.3. Unit Of Work pattern 353
      12.3.3.1. Several Batches 353
      12.3.3.2. Updating only the mapped fields 353
   12.3.4. Local network as bottleneck 355
      12.3.4.1. Array binding 356
         12.3.4.1.1. For faster BATCH mode 356
         12.3.4.1.2. For faster IN clause 357
      12.3.4.2. Optimized SQL for bulk insert 357
12.4. CRUD level cache 359
   12.4.1. Where to cache 359
   12.4.2. When to cache 360
   12.4.3. What to cache 360
   12.4.4. How to cache 361
   12.4.5. Business logic and API cache 362

13. Server side SQL/ORM process

13.1. Optimize for performance 364
   13.1.1. Profiling your application 364
   13.1.2. Client-side caching 364
13.1.3. Write business logic as Services 365
13.1.4. Using ORM full power 365

13.2. Stored procedures 366
13.2.1. Why to avoid stored procedures 366
13.2.2. Stored procedures, anyway 366
  13.2.2.1. Custom SQL functions 367
    13.2.2.1.1. Implementing a function 367
    13.2.2.1.2. Registering a function 368
      13.2.2.1.2.1. Direct low-level SQLite3 registration 368
      13.2.2.1.2.2. Class-driven registration 368
      13.2.2.1.2.3. Custom class definition 369
    13.2.2.2. Low-level SQLite3 stored procedure in Delphi 371
  13.2.2.3. External stored procedure 371

13.3. Server side Services 372

14. Client-Server services via methods

| 14.1. Publishing a service on the server | 373 |
| 14.2. Defining the client | 375 |
| 14.3. Direct parameter marshalling on server side | 376 |
| 14.4. Returns non-JSON content | 377 |
| 14.5. Advanced process on server side | 378 |
| 14.6. Browser speed-up for unmodified requests | 379 |
| 14.7. Returning file content | 379 |
| 14.8. JSON Web Tokens (JWT) | 379 |
| 14.9. Handling errors | 383 |
| 14.10. Benefits and limitations of this implementation | 384 |

15. Interfaces

| 15.1. Delphi and interfaces | 385 |
| 15.1.1. Declaring an interface | 385 |
| 15.1.2. Implementing an interface with a class | 386 |
15.1.3. Using an interface

15.1.4. There is more than one way to do it

15.1.5. Here comes the magic

15.2. SOLID design principles

15.2.1. Single Responsibility Principle

15.2.1.1. Splitting classes

15.2.1.2. Do not mix UI and logic

15.2.2. Open/Closed Principle

15.2.2.1. Applied to our framework units

15.2.2.2. Open/Closed in practice

15.2.2.3. No Singleton nor global variables

15.2.3. Liskov Substitution Principle

15.2.3.1. Use parent classes

15.2.3.2. I'm your father, Luke

15.2.3.3. Don't check the type at runtime

15.2.3.4. Partially abstract classes

15.2.3.5. Messing units dependencies

15.2.3.6. Practical advantages

15.2.4. Interface Segregation Principle

15.2.4.1. Consequence of the other principles

15.2.4.2. Using interfaces

15.2.5. Dependency Inversion Principle

15.2.5.1. Upside Down Development

15.2.5.2. Injection patterns

15.3. Circular reference and (zeroing) weak pointers

15.3.1. Weak pointers

15.3.2. Handling weak pointers

15.3.3. Zeroing weak pointers

15.3.4. Weak pointers functions implementation details

15.4. Interfaces in practice: dependency injection, stubs and mocks
15.4.1. Dependency Injection at constructors

15.4.2. Why use fake / emulated interfaces?
- 15.4.2.1. Stubs and mocks
- 15.4.2.2. Defining stubs
- 15.4.2.3. Defining a mock
- 15.4.2.4. Running the test

15.5. Stubs and Mocks in mORMot
- 15.5.1. Direct use of interface types without TypeInfo()
- 15.5.2. Manual dependency injection
- 15.5.3. Stubbing complex return values
- 15.5.4. Stubbing via a custom delegate or callback
  - 15.5.4.1. Delegate with named variant parameters
  - 15.5.4.2. Delegate with indexed variant parameters
  - 15.5.4.3. Delegate with JSON parameters
  - 15.5.4.4. Accessing the test case when mocking
- 15.5.5. Calls tracing

15.6. Dependency Injection and Interface Resolution

16. Client-Server services via interfaces
- 16.1. Implemented features
- 16.2. How to make services
- 16.3. Defining a service contract
  - 16.3.1. Define an interface
  - 16.3.2. Service Methods Parameters
  - 16.3.3. TPersistent / TSQLRecord parameters
  - 16.3.4. Record parameters
  - 16.3.5. TCollection parameters
    - 16.3.5.1. Use of TCollection
    - 16.3.5.2. Inherit from TInterfacedCollection
    - 16.3.5.3. Register a TCollection type
- 16.4. Server side
16.4.1. Implementing the service contract 429
16.4.2. Set up the Server factory 429
16.4.3. Instances life time implementation 429
16.4.4. Accessing low-level execution context 432
  16.4.4.1. Retrieve information from the global ServiceContext 432
  16.4.4.2. Implement your service from TinjectableObjectRest 433
16.4.5. Using services on the Server side 433
16.5. Client side 435
  16.5.1. Set up the Client factory 435
  16.5.2. Using services on the Client side 435
16.6. Sample code 437
  16.6.1. The shared contract 437
  16.6.2. The server sample application 437
  16.6.3. The client sample application 438
  16.6.4. Enhanced sample: remote SQL access 439
16.7. Asynchronous callbacks 444
  16.7.1. WebSockets support 444
    16.7.1.1. Using a "Saga" callback to notify long term end-of-process 446
    16.7.1.2. Client service consumption 446
    16.7.1.3. Server side implementation 447
  16.7.2. Publish-subscribe for events 449
    16.7.2.1. Defining the interfaces 449
    16.7.2.2. Writing the Publisher 450
    16.7.2.3. Consuming the service from the Subscriber side 451
    16.7.2.4. Subscriber multiple redirection 452
    16.7.2.5. Proper threaded implementation 453
    16.7.2.6. Interacting with UI/VCL 453
  16.7.3. Interface callbacks instead of class messages 454
    16.7.3.1. Using service and callback interfaces 454
16.7.3.2. Classical message(s) event
16.7.3.3. Workflow adaptation
16.7.3.4. From interfaces comes abstraction and ease

16.8. Implementation details

16.8.1. Error handling
16.8.2. Security
16.8.3. Implementation class types
16.8.4. Server-side execution options (threading)

16.8.5. Audit Trail for Services
16.8.5.1. When logging is not enough
16.8.5.2. Tracing Service Methods
16.8.5.3. Tracing Asynchronous External Calls

16.8.6. Transmission content
16.8.6.1. Request format
16.8.6.1.1. REST mode
16.8.6.1.1.1. Parameters transmitted as JSON array
16.8.6.1.1.2. Parameters transmitted as JSON object
16.8.6.1.1.3. Parameters encoded at URI level
16.8.6.1.1.4. Sending a JSON object
16.8.6.1.1.5. Sending raw binary
16.8.6.1.2. JSON-RPC
16.8.6.1.2.1. Parameters transmitted as JSON array
16.8.6.1.2.2. Parameters transmitted as JSON object
16.8.6.1.3. REST mode or JSON-RPC mode?
16.8.6.2. Response format
16.8.6.2.1. Standard answer as JSON object
16.8.6.2.1.1. JSON answers
16.8.6.2.1.1.1. Returning as JSON array
16.8.6.2.1.1.2. Returning a JSON object
16.8.6.2.1.2. Returning raw JSON content
16.8.6.2.1.3. Returning errors
16.8.6.2.1.2. Returning raw JSON content
16.8.6.2.1.3. Returning errors
16.8.6.2.2. Returning content as XML 474
16.8.6.2.2.1. Always return XML content 474
16.8.6.2.2.2. Return XML content on demand 475
16.8.6.2.3. Custom returned content 476

16.9. Comparison with WCF 478

17. Cross-Platform clients

17.1. Available client platforms 483
17.1.1. Delphi FMX / FreePascal FCL cross-platform support 483
17.1.1.1. Cross-platform JSON 483
17.1.1.2. Delphi OSX and NextGen 485
17.1.1.3. FreePascal clients 485
17.1.1.4. Local or remote logging 486

17.1.2. Smart Mobile Studio support 486
17.1.2.1. Beyond JavaScript 487
17.1.2.2. Using Smart Mobile Studio with mORMot 487

17.1.3. Remote logging 488

17.2. Generating client wrappers 489
17.2.1. Publishing the code generator 489
17.2.2. Delphi / FreePascal client samples 491
17.2.2.1. Connection to the server 492
17.2.2.2. CRUD/ORM remote access 492
17.2.2.3. Service consumption 494

17.2.3. Smart Mobile Studio client samples 496
17.2.3.1. Adding two numbers in AJAX 496
17.2.3.2. CRUD/ORM remote access 499

18. MVC pattern

18.1. Model 502
18.2. Views 502
18.2.1. Desktop clients 503
18.2.1.1. RTTI 503
18.2.1.2. User Interface 504

18.2.2. Web clients 505
   18.2.2.1. Mustache template engine 505
   18.2.2.2. Mustache principles 505
   18.2.2.3. Mustache templates 506
      18.2.2.3.1. Variables 508
      18.2.2.3.2. Sections 508
      18.2.2.3.3. Inverted Sections 510
      18.2.2.3.4. Partials 510
   18.2.2.4. SynMustache unit 511
      18.2.2.4.1. Variables 512
      18.2.2.4.2. Sections 513
      18.2.2.4.3. Partials 513
      18.2.2.4.4. Expression Helpers 514
      18.2.2.4.5. Internationalization 515
   18.2.2.5. Low-level integration with method-based services 516
   18.2.2.6. MVC/MVVM Design 516

19. MVC/MVVM Web applications 519

19.1. MVCModel 519
   19.1.1. Data Model 519
   19.1.2. Hosted in a REST server over HTTP 521

19.2. MVCViewModel 522
   19.2.1. Defining the commands 522
   19.2.2. Implementing the Controller 526
   19.2.3. Variable input parameters 528
   19.2.4. Using Services in the Controller 529
   19.2.5. Controller Thread Safety 529
   19.2.6. Web Sessions 530

19.3. Writing the Views 533

20. Hosting
20.1. Windows and Linux hosted

20.2. Deployment Architecture
   20.2.1. Shared server
   20.2.2. Two servers
   20.2.3. Two instances on the same server
   20.2.4. Scaling via CDN

21. Security
   21.1. Authentication
      21.1.1. Principles
         21.1.1.1. HTTP basic auth over HTTPS
         21.1.1.2. Session via Cookies
         21.1.1.3. Query Authentication
      21.1.2. Framework authentication
         21.1.2.1. Per-User authentication
         21.1.2.2. Session handling
      21.1.3. Authentication schemes
         21.1.3.1. Class-driven authentication
         21.1.3.2. mORMot secure RESTful authentication
         21.1.3.3. Authentication using Windows credentials
            21.1.3.3.1. Windows Authentication
            21.1.3.3.2. Using NTLM or Kerberos
         21.1.3.4. Weak authentication
         21.1.3.5. HTTP Basic authentication
      21.1.4. Clients authentication
         21.1.4.1. Client interactivity
         21.1.4.2. Authentication using AJAX
      21.1.5. JWT Authentication
   21.2. Authorization
      21.2.1. Per-table access rights
      21.2.2. Additional safety
         21.2.2.1. SQL remote execution
21.2.2. Service execution

22. Scripting Engine

22.1. Scripting abilities

22.2. SpiderMonkey integration

22.2.1. A powerful JavaScript engine

22.2.2. Direct access to the SpiderMonkey API

22.2.3. Execution scheme

22.2.4. Creating your execution context

22.2.5. Blocking threading model

22.3. Interaction with existing code

22.3.1. Proper engine initialization

22.3.2. Calling Delphi code from JavaScript

22.3.3. TSMVariant custom type

22.3.4. Calling JavaScript code from Delphi

23. Asymmetric Encryption

23.1. Public-key Cryptography

23.1.1. Keys Generation and Distribution

23.1.2. Message Authentication

23.1.3. Certificates and Public Key Infrastructure

23.1.4. Message encryption

23.2. Elliptic Curve Cryptography

23.2.1. Introducing SynEcc

23.2.2. ECC command line tool

23.2.3. Keys and Certificates Generation

23.2.4. TECCCertificate and TECCCertificateSecret

23.2.5. File Signature

23.2.6. Signing in Code

23.2.7. File Encryption

23.2.8. Private Keys Passwords Cheat Mode

23.2.9. Encryption in Code
23.3. Application Locking

23.3.1. From the User perspective

23.3.2. From the Support Team perspective

23.3.3. Benefits of Asymmetric Encryption for License management

24. Domain-Driven-Design

24.1. Domain

24.2. Modeling

24.2.1. Several Models to rule them all

24.2.2. The state of the model

24.2.3. Composition

24.3. DDD model

24.3.1. Ubiquitous Language

24.3.2. Value Objects and Entities

24.3.3. Aggregates

24.3.4. Factory and Repository patterns

24.3.5. DTO and Events to avoid domain leaking

24.3.6. Services

24.3.7. Clean Uncoupled Architecture

24.4. mORMot’s DDD

24.4.1. Designer’s commitments

24.4.2. Defining objects in Delphi

24.4.3. Defining DDD objects in mORMot

24.4.3.1. Use framework types for DDD objects

24.4.3.2. Define uncoupled DDD objects

24.4.3.3. Specialize your simple types

24.4.3.4. Define your PODO classes

24.4.3.5. Store your Entities in CQRS Repositories

24.4.3.5.1. CQRS Interfaces

24.4.3.5.2. Queries Interface

24.4.3.5.3. Commands Interface
24.4.3.5.4. Automated Repository using the ORM 614

24.4.3.5.4.1. DDD / ORM mapping 614
24.4.3.5.4.2. Define the Factory 616
24.4.3.5.4.3. Implement the CQRS methods 618

24.4.3.6. Isolate your Domain using DTOs 620

24.4.4. Defining services 620

24.4.5. Event-Driven Design 621

24.4.5.1. Events as Callbacks 621
24.4.5.2. Event Sourcing via Event Oriented Databases 622

24.4.6. Building a Clean architecture 623

25. Testing and logging

25.1. Automated testing 627

25.1.1. Involved classes in Unitary testing 627
25.1.2. First steps in testing 628
25.1.3. Framework test coverage 629

25.2. Enhanced logging 631

25.2.1. Setup logging 631
25.2.2. Call trace 633
25.2.3. Including symbol definitions 633
25.2.4. Exception handling 633
25.2.5. Serialization 635
25.2.6. Multi-threaded applications 635
25.2.7. Log to the console 636
25.2.8. Remote logging 636
25.2.9. Log to third-party libraries 637
25.2.10. Automated log archival 637
25.2.11. Log files rotation 638
25.2.12. Integration within tests 639
25.2.13. Log Viewer 639

25.2.13.1. Open log files 639
25.2.13.2. Log browser 640
25.2.13.3. Customer-side profiler 640
25.2.13.4. Per-thread inspection 641
25.2.13.5. Server for remote logging 641
25.2.14. Framework log integration 641

26. Source code

26.1. License 643
   26.1.1. Three Licenses Model 643
   26.1.2. Publish modifications and credit for the library 644
   26.1.3. Derivate Open Source works 644
   26.1.4. Commercial licenses 645

26.2. Availability 646
   26.2.1. Obtaining the Source Code 646
   26.2.2. Expected compilation platform 646
   26.2.3. SQLite3 static linking for Delphi and FPC 647
   26.2.4. SpiderMonkey library 648
   26.2.5. Folder layout 648
      26.2.5.1. Root folder 649
      26.2.5.2. SynDBDataset folder 651
      26.2.5.3. SQLite3 folder 651
      26.2.5.4. CrossPlatform folder 652

26.3. Delphi Installation 653
   26.3.1. Manual download 653
   26.3.2. Get from GitHub 653
   26.3.3. Setup the Delphi IDE 653

26.4. FreePascal / Lazarus Installation 655
   26.4.1. Possible targets 655
   26.4.2. Setup your dedicated FPC / Lazarus environment with fpcupdeluxe 655
   26.4.3. Missing RTTI for interfaces in old FPC 2.6 656
   26.4.4. Writing your project for FPC 657
   26.4.5. Linux VM installation tips 658
26.5. CrossKylix support

26.5.1. What is Cross-Kylix?

26.5.2. Running Kylix 32-bit executables on 64-bit Linux

26.6. Upgrading from a 1.17 revision

27. mORMot Framework source

27.1. mORMot Framework used Units

27.2. PasZip.pas unit

27.3. SynBidirSock.pas unit

27.4. SynBigTable.pas unit

27.5. SynCommons.pas unit

27.6. SynCrtSock.pas unit

27.7. SynCrypto.pas unit

27.8. SynCurl.pas unit

27.9. SynDB.pas unit

27.10. SynDBDataset.pas unit

27.11. SynDBMidasVCL.pas unit

27.12. SynDBODBC.pas unit

27.13. SynDBOracle.pas unit

27.14. SynDBPostgres.pas unit

27.15. SynDBRemote.pas unit

27.16. SynDBSQLite3.pas unit

27.17. SynDBVCL.pas unit

27.18. SynDBZeos.pas unit

27.19. SynEcc.pas unit

27.20. SynFastWideString.pas unit

27.21. SynGdiPlus.pas unit

27.22. SynLizard.pas unit

27.23. SynLog.pas unit

27.24. SynLZ.pas unit
27.25. SynLZO.pas unit 1395
27.26. SynMongoDB.pas unit 1396
27.27. SynMustache.pas unit 1450
27.28. SynOleDB.pas unit 1458
27.29. SynPdf.pas unit 1473
27.30. SynProtoRelay.pas unit 1521
27.31. SynProtoRTSPHTTP.pas unit 1526
27.32. SynSelfTests.pas unit 1528
27.33. SynSM.pas unit 1554
27.34. SynSMAPI.pas unit 1573
27.35. SynSQLite3.pas unit 1646
27.36. SynSQLite3RegEx.pas unit 1710
27.37. SynSQLite3Static.pas unit 1711
27.38. SynSSPI.pas unit 1714
27.39. SynSSPIAuth.pas unit 1718
27.40. SynTable.pas unit 1721
27.41. SynTaskDialog.pas unit 1824
27.42. SynTests.pas unit 1832
27.43. SynVirtualDataSet.pas unit 1840
27.44. SynWinSock.pas unit 1843
27.45. SynZip.pas unit 1845
27.46. SynZipFiles.pas unit 1856
27.47. SynCrossPlatformCrypto.pas unit 1859
27.48. SynCrossPlatformJSON.pas unit 1861
27.49. SynCrossPlatformREST.pas unit 1870
27.50. SynCrossPlatformSpecific.pas unit 1892
27.51. SynCrossPlatformTests.pas unit 1897
27.52. mORMot.pas unit 1899
27.53. mORMotDB.pas unit 2276
| 27.54. | mORMotDDD.pas unit | 2285 |
| 27.55. | mORMotFastCgiServer.pas unit | 2302 |
| 27.56. | mORMotHttpClient.pas unit | 2306 |
| 27.57. | mORMotHttpServer.pas unit | 2313 |
| 27.58. | mORMoti18n.pas unit | 2322 |
| 27.59. | mORMotMidasVCL.pas unit | 2332 |
| 27.60. | mORMotMongoDB.pas unit | 2334 |
| 27.61. | mORMotMVC.pas unit | 2339 |
| 27.62. | mORMotReport.pas unit | 2352 |
| 27.63. | mORMotSelfTests.pas unit | 2369 |
| 27.64. | mORMotService.pas unit | 2370 |
| 27.65. | mORMotSQLite3.pas unit | 2382 |
| 27.66. | mORMotToolBar.pas unit | 2390 |
| 27.67. | mORMotUI.pas unit | 2404 |
| 27.68. | mORMotUIEdit.pas unit | 2415 |
| 27.69. | mORMotUILogin.pas unit | 2418 |
| 27.70. | mORMotUIOptions.pas unit | 2422 |
| 27.71. | mORMotUIQuery.pas unit | 2424 |
| 27.72. | mORMotVCL.pas unit | 2426 |
| 27.73. | mORMotWrappers.pas unit | 2430 |
| 27.74. | TestSQL3FPCInterfaces.pas unit | 2436 |
| 27.75. | dddDomAuthInterfaces.pas unit | 2437 |
| 27.76. | dddDomCountry.pas unit | 2439 |
| 27.77. | dddDomUserCQRS.pas unit | 2442 |
| 27.78. | dddDomUserInterfaces.pas unit | 2445 |
| 27.79. | dddDomUserTypes.pas unit | 2447 |
| 27.80. | dddInfraApps.pas unit | 2450 |
| 27.81. | dddInfraAuthRest.pas unit | 2463 |
| 27.82. | dddInfraEmail.pas unit | 2467 |
27.83. dddInfraEmailer.pas unit  
27.84. dddInfraRepoUser.pas unit  
27.85. dddInfraSettings.pas unit  
27.86. SynDBBDE.pas unit  
27.87. SynDBFireDAC.pas unit  
27.88. SynDBNexusDB.pas unit  
27.89. SynDBUniDAC.pas unit  

28. SynFile application  
28.1. General architecture  
28.2. Database design  
28.3. Client Server implementation  
28.4. User Interface generation  
28.4.1. Rendering  
28.4.2. Enumeration types  
28.4.3. ORM Registration  
28.4.4. Main window  
28.5. Report generation  
28.6. Application i18n and L10n  
28.6.1. Creating the reference file  
28.6.2. Adding a new language  
28.6.3. Language selection  
28.6.4. Manual translation  
28.6.5. TForm / TFrame hook  
28.6.6. Localization  

29. Main SynFile Demo source  
29.1. Main SynFile Demo used Units  
29.2. FileClient.pas unit  
29.3. FileEdit.pas unit  
29.4. FileMain.pas unit
### 29.5. FileServer.pas unit
- Page 2536

### 29.6. FileTables.pas unit
- Page 2538

### 30. SWRS implications

#### Software Architecture Design Reference Table
- Page 2542

#### 30.1. Client Server ORM/SOA framework
- Page 2543

- 30.1.1. SWRS # DI-2.1.1
- Page 2543

- 30.1.2. SWRS # DI-2.1.1.1
- Page 2543

- 30.1.3. SWRS # DI-2.1.1.2.1
- Page 2544

- 30.1.4. SWRS # DI-2.1.1.2.2
- Page 2544

- 30.1.5. SWRS # DI-2.1.1.2.3
- Page 2544

- 30.1.6. SWRS # DI-2.1.1.2.4
- Page 2545

- 30.1.7. SWRS # DI-2.1.2
- Page 2545

- 30.1.8. SWRS # DI-2.1.3
- Page 2546

- 30.1.9. SWRS # DI-2.1.4
- Page 2547

- 30.1.10. SWRS # DI-2.1.5
- Page 2547

#### 30.2. SQlite3 engine
- Page 2548

- 30.2.1. SWRS # DI-2.2.1
- Page 2548

- 30.2.2. SWRS # DI-2.2.2
- Page 2549

#### 30.3. User interface
- Page 2549

- 30.3.1. SWRS # DI-2.3.1.1
- Page 2549

- 30.3.2. SWRS # DI-2.3.1.2
- Page 2549

- 30.3.3. SWRS # DI-2.3.1.3
- Page 2550

- 30.3.4. SWRS # DI-2.3.2
- Page 2550
**Pictures Reference Table**

The following table is a quick-reference guide to all the Pictures referenced in this *Software Architecture Design* (SAD) document.

<table>
<thead>
<tr>
<th>Pictures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adopt a mORMot</td>
<td>104, 341, 519, 2505</td>
</tr>
<tr>
<td>Adopt a mORMot</td>
<td>130, 363, 536</td>
</tr>
<tr>
<td>Adopt a mORMot</td>
<td>188, 372, 544</td>
</tr>
<tr>
<td>Adopt a mORMot</td>
<td>196, 373, 561</td>
</tr>
<tr>
<td>Adopt a mORMot</td>
<td>239, 385, 571</td>
</tr>
<tr>
<td>Adopt a mORMot</td>
<td>278, 419, 595</td>
</tr>
<tr>
<td>Adopt a mORMot</td>
<td>295, 481, 626</td>
</tr>
<tr>
<td>Adopt a mORMot</td>
<td>81, 318, 502, 643</td>
</tr>
<tr>
<td>Client-Server implementation - Client side</td>
<td>342</td>
</tr>
<tr>
<td>Domain Driven Design n-Tier Architecture - Physical View</td>
<td>89</td>
</tr>
<tr>
<td>Domain-Driven Design - Building Blocks</td>
<td>99, 598</td>
</tr>
<tr>
<td>Domain Driven Design n-Tier Architecture - Logical View</td>
<td>89</td>
</tr>
<tr>
<td>Architecture Iterative Process (SCRUM)</td>
<td>84</td>
</tr>
<tr>
<td>Multi-Tier Architecture - Logical View</td>
<td>88</td>
</tr>
<tr>
<td>Service Oriented Architecture - Logical View</td>
<td>90</td>
</tr>
<tr>
<td>Service Oriented Architecture - Logical View of Composition</td>
<td>90</td>
</tr>
<tr>
<td>Strings in Domain Driven Design n-Tier Architecture</td>
<td>136</td>
</tr>
<tr>
<td>Two-Tier Architecture - Logical View</td>
<td>88</td>
</tr>
<tr>
<td>Client-Server implementation - Server side</td>
<td>342</td>
</tr>
<tr>
<td>Client-Server implementation - Server side</td>
<td>346</td>
</tr>
<tr>
<td>Client-Server implementation - Server side with Virtual Tables</td>
<td>344</td>
</tr>
<tr>
<td>Client-Server implementation - Stand-Alone application</td>
<td>343</td>
</tr>
<tr>
<td>Client-Server implementation - Server side with &quot;static&quot; Virtual Tables</td>
<td>345</td>
</tr>
<tr>
<td>Asymmetric Encryption Scheme</td>
<td>574</td>
</tr>
<tr>
<td>Asymmetric Key Generation</td>
<td>572</td>
</tr>
<tr>
<td>Pictures</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Asymmetric Digital Signature</td>
<td>573</td>
</tr>
<tr>
<td>Asymmetric Sign-Then-Encrypt Scheme</td>
<td>575</td>
</tr>
<tr>
<td>BATCH mode Client-Server latency</td>
<td>350</td>
</tr>
<tr>
<td>BATCH mode latency issue on external DB</td>
<td>355</td>
</tr>
<tr>
<td>HTTP/1.1 Client architecture</td>
<td>330</td>
</tr>
<tr>
<td>RESTful Client classes</td>
<td>323</td>
</tr>
<tr>
<td>HTTP/1.1 Client RESTful classes</td>
<td>329</td>
</tr>
<tr>
<td>RESTful Client-Server classes</td>
<td>321</td>
</tr>
<tr>
<td>AuditTrail Record Layout</td>
<td>2509</td>
</tr>
<tr>
<td>AuthGroup Record Layout</td>
<td>548</td>
</tr>
<tr>
<td>AuthUser Record Layout</td>
<td>549</td>
</tr>
<tr>
<td>Data Record Layout</td>
<td>2508</td>
</tr>
<tr>
<td>History Record Layout</td>
<td>179</td>
</tr>
<tr>
<td>Memo Record Layout</td>
<td>2508</td>
</tr>
<tr>
<td>SafeData Record Layout</td>
<td>2509</td>
</tr>
<tr>
<td>SafeMemo Record Layout</td>
<td>2508</td>
</tr>
<tr>
<td>ServiceLog Record Layout</td>
<td>464</td>
</tr>
<tr>
<td>ServiceNotifications Record Layout</td>
<td>465</td>
</tr>
<tr>
<td>CQRS Repository Service Interface for TUser</td>
<td>611</td>
</tr>
<tr>
<td>CQRS Dogmatic Repository Service Interface for TUser</td>
<td>611</td>
</tr>
<tr>
<td>CQRS Class Hierarchy Mapping for ORM and DDD Entities</td>
<td>615</td>
</tr>
<tr>
<td>Application Unlocking via Asymmetric Cryptography</td>
<td>591</td>
</tr>
<tr>
<td>Application Unlocking on Two Computers</td>
<td>592</td>
</tr>
<tr>
<td>Application Unlocking for Two Users</td>
<td>592</td>
</tr>
<tr>
<td>Design Inputs, FMEA and Risk Specifications</td>
<td>59</td>
</tr>
<tr>
<td>FTS ORM classes</td>
<td>216</td>
</tr>
<tr>
<td>ESQLQueryException classes hierarchy</td>
<td>247</td>
</tr>
<tr>
<td>External Databases classes hierarchy</td>
<td>273</td>
</tr>
<tr>
<td>SynFile TSQLRecord classes hierarchy</td>
<td>2507</td>
</tr>
<tr>
<td>Pictures</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TJWTAbstract classes hierarchy</td>
<td>380</td>
</tr>
<tr>
<td>TOleDbStatement classes hierarchy</td>
<td>247</td>
</tr>
<tr>
<td>TSQlDataBaseSQLFunction classes hierarchy</td>
<td>369</td>
</tr>
<tr>
<td>TSQlDBConnectionProperties classes hierarchy</td>
<td>246</td>
</tr>
<tr>
<td>SynDB Remote access Server classes hierarchy</td>
<td>263</td>
</tr>
<tr>
<td>TSQlDBSQLite3Connection classes hierarchy</td>
<td>246</td>
</tr>
<tr>
<td>SynDB Remote access Client classes hierarchy</td>
<td>264</td>
</tr>
<tr>
<td>TSQlite3Library classes hierarchy</td>
<td>209</td>
</tr>
<tr>
<td>Custom Virtual Tables records classes hierarchy</td>
<td>229</td>
</tr>
<tr>
<td>TSQlRestClient classes hierarchy</td>
<td>313</td>
</tr>
<tr>
<td>TSQlRestServerAuthentication classes hierarchy</td>
<td>551</td>
</tr>
<tr>
<td>Routing via TSQlRestServerURIContext classes hierarchy</td>
<td>466</td>
</tr>
<tr>
<td>Virtual Tables classes hierarchy</td>
<td>225</td>
</tr>
<tr>
<td>Filtering and Validation classes hierarchy</td>
<td>172</td>
</tr>
<tr>
<td>Default filters and Validation classes hierarchy</td>
<td>173</td>
</tr>
<tr>
<td>TSynTest classes hierarchy</td>
<td>627</td>
</tr>
<tr>
<td>MongoDB TSynDirect classes hierarchy</td>
<td>285</td>
</tr>
<tr>
<td>TWebSocketProtocolJSON classes hierarchy</td>
<td>445</td>
</tr>
<tr>
<td>Corporate Servers Redirection</td>
<td>235</td>
</tr>
<tr>
<td>THttpServerGeneric classes hierarchy</td>
<td>325</td>
</tr>
<tr>
<td>SynCrossPlatformCrypto class hierarchy</td>
<td>1859</td>
</tr>
<tr>
<td>SynCrossPlatformJSON class hierarchy</td>
<td>1861</td>
</tr>
<tr>
<td>SynCrossPlatformREST class hierarchy</td>
<td>1870</td>
</tr>
<tr>
<td>SynCrossPlatformSpecific class hierarchy</td>
<td>1892</td>
</tr>
<tr>
<td>SynCrossPlatformTests class hierarchy</td>
<td>1897</td>
</tr>
<tr>
<td>PasZip class hierarchy</td>
<td>674</td>
</tr>
<tr>
<td>dddDomAuthInterfaces class hierarchy</td>
<td>2437</td>
</tr>
<tr>
<td>dddDomCountry class hierarchy</td>
<td>2439</td>
</tr>
<tr>
<td>dddDomUserCQRS class hierarchy</td>
<td>2442</td>
</tr>
<tr>
<td>Pictures</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>dddDomUserInterfaces class hierarchy</td>
<td>2445</td>
</tr>
<tr>
<td>dddDomUserTypes class hierarchy</td>
<td>2447</td>
</tr>
<tr>
<td>dddInfraApps class hierarchy</td>
<td>2451</td>
</tr>
<tr>
<td>dddInfraAuthRest class hierarchy</td>
<td>2463</td>
</tr>
<tr>
<td>dddInfraEmail class hierarchy</td>
<td>2468</td>
</tr>
<tr>
<td>dddInfraEmailer class hierarchy</td>
<td>2472</td>
</tr>
<tr>
<td>dddInfraRepoUser class hierarchy</td>
<td>2475</td>
</tr>
<tr>
<td>dddInfraSettings class hierarchy</td>
<td>2478</td>
</tr>
<tr>
<td>mORMot class hierarchy</td>
<td>1901</td>
</tr>
<tr>
<td>mORMotDB class hierarchy</td>
<td>2276</td>
</tr>
<tr>
<td>mORMotDDD class hierarchy</td>
<td>2286</td>
</tr>
<tr>
<td>mORMotFastCgiServer class hierarchy</td>
<td>2302</td>
</tr>
<tr>
<td>mORMotHttpClient class hierarchy</td>
<td>2307</td>
</tr>
<tr>
<td>mORMotHttpServer class hierarchy</td>
<td>2314</td>
</tr>
<tr>
<td>mORMoti18n class hierarchy</td>
<td>2322</td>
</tr>
<tr>
<td>mORMot MongoDB class hierarchy</td>
<td>2334</td>
</tr>
<tr>
<td>mORMotMVC class hierarchy</td>
<td>2340</td>
</tr>
<tr>
<td>mORMotReport class hierarchy</td>
<td>2352</td>
</tr>
<tr>
<td>mORMotService class hierarchy</td>
<td>2370</td>
</tr>
<tr>
<td>mORMotSQLite3 class hierarchy</td>
<td>2382</td>
</tr>
<tr>
<td>mORMotToolBar class hierarchy</td>
<td>2391</td>
</tr>
<tr>
<td>mORMotUI class hierarchy</td>
<td>2404</td>
</tr>
<tr>
<td>mORMotUIEdit class hierarchy</td>
<td>2415</td>
</tr>
<tr>
<td>mORMotUILogin class hierarchy</td>
<td>2418</td>
</tr>
<tr>
<td>mORMotUIOptions class hierarchy</td>
<td>2422</td>
</tr>
<tr>
<td>mORMotUIQuery class hierarchy</td>
<td>2424</td>
</tr>
<tr>
<td>mORMotVCL class hierarchy</td>
<td>2426</td>
</tr>
<tr>
<td>FileClient class hierarchy</td>
<td>2530</td>
</tr>
<tr>
<td>FileEdit class hierarchy</td>
<td>2532</td>
</tr>
<tr>
<td>Class Hierarchy</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>FileMain class hierarchy</td>
<td>2535</td>
</tr>
<tr>
<td>FileServer class hierarchy</td>
<td>2536</td>
</tr>
<tr>
<td>FileTables class hierarchy</td>
<td>2538</td>
</tr>
<tr>
<td>SynBidirSock class hierarchy</td>
<td>681</td>
</tr>
<tr>
<td>SynBigTable class hierarchy</td>
<td>704</td>
</tr>
<tr>
<td>SynCommons class hierarchy</td>
<td>718</td>
</tr>
<tr>
<td>SynCrtSock class hierarchy</td>
<td>1083</td>
</tr>
<tr>
<td>SynCrypto class hierarchy</td>
<td>1140</td>
</tr>
<tr>
<td>SynCurl class hierarchy</td>
<td>1200</td>
</tr>
<tr>
<td>SynDB class hierarchy</td>
<td>1204</td>
</tr>
<tr>
<td>SynDBBDE class hierarchy</td>
<td>2490</td>
</tr>
<tr>
<td>SynDBFireDAC class hierarchy</td>
<td>2493</td>
</tr>
<tr>
<td>SynDBNexusDB class hierarchy</td>
<td>2497</td>
</tr>
<tr>
<td>SynDBUniDAC class hierarchy</td>
<td>2501</td>
</tr>
<tr>
<td>SynDBDataset class hierarchy</td>
<td>1267</td>
</tr>
<tr>
<td>SynDBMidasVCL class hierarchy</td>
<td>1271</td>
</tr>
<tr>
<td>SynDBODBC class hierarchy</td>
<td>1274</td>
</tr>
<tr>
<td>SynDBOracle class hierarchy</td>
<td>1280</td>
</tr>
<tr>
<td>SynDBPostgres class hierarchy</td>
<td>1289</td>
</tr>
<tr>
<td>SynDBRemote class hierarchy</td>
<td>1293</td>
</tr>
<tr>
<td>SynDBSQLite3 class hierarchy</td>
<td>1298</td>
</tr>
<tr>
<td>SynDBVCL class hierarchy</td>
<td>1304</td>
</tr>
<tr>
<td>SynDBZeos class hierarchy</td>
<td>1307</td>
</tr>
<tr>
<td>SynEcc class hierarchy</td>
<td>1313</td>
</tr>
<tr>
<td>SynGdiPlus class hierarchy</td>
<td>1350</td>
</tr>
<tr>
<td>SynLizard class hierarchy</td>
<td>1359</td>
</tr>
<tr>
<td>SynLog class hierarchy</td>
<td>1363</td>
</tr>
<tr>
<td>SynMongoDB class hierarchy</td>
<td>1397</td>
</tr>
<tr>
<td>SynMustache class hierarchy</td>
<td>1450</td>
</tr>
<tr>
<td>Pictures</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>SynOleDb class hierarchy</td>
<td>1459</td>
</tr>
<tr>
<td>SynPdf class hierarchy</td>
<td>1474</td>
</tr>
<tr>
<td>SynProtoRelay class hierarchy</td>
<td>1522</td>
</tr>
<tr>
<td>SynProtoRTSPHTTP class hierarchy</td>
<td>1526</td>
</tr>
<tr>
<td>SynSelfTests class hierarchy</td>
<td>1532</td>
</tr>
<tr>
<td>SynSM class hierarchy</td>
<td>1554</td>
</tr>
<tr>
<td>SynSMAPI class hierarchy</td>
<td>1573</td>
</tr>
<tr>
<td>SynSQLite3 class hierarchy</td>
<td>1646</td>
</tr>
<tr>
<td>SynSQLite3Static class hierarchy</td>
<td>1711</td>
</tr>
<tr>
<td>SynSSPI class hierarchy</td>
<td>1714</td>
</tr>
<tr>
<td>SynTable class hierarchy</td>
<td>1722</td>
</tr>
<tr>
<td>SynTaskDialog class hierarchy</td>
<td>1824</td>
</tr>
<tr>
<td>SynTests class hierarchy</td>
<td>1832</td>
</tr>
<tr>
<td>SynVirtualDataSet class hierarchy</td>
<td>1840</td>
</tr>
<tr>
<td>SynWinSock class hierarchy</td>
<td>1843</td>
</tr>
<tr>
<td>SynZip class hierarchy</td>
<td>1845</td>
</tr>
<tr>
<td>SynZipFiles class hierarchy</td>
<td>1856</td>
</tr>
<tr>
<td>CRUD caching in mORMot</td>
<td>359</td>
</tr>
<tr>
<td>mORMot Persistence Layer Architecture</td>
<td>196, 239, 278</td>
</tr>
<tr>
<td>Clean Domain-Oriented Architecture of mORMot</td>
<td>624</td>
</tr>
<tr>
<td>Alternate Domain-Oriented Architecture of mORMot</td>
<td>625</td>
</tr>
<tr>
<td>General mORMot architecture - Client / Server</td>
<td>74</td>
</tr>
<tr>
<td>General mORMot architecture - Stand-alone application</td>
<td>74</td>
</tr>
<tr>
<td>General mORMot architecture - Client Server implementation</td>
<td>82</td>
</tr>
<tr>
<td>General mORMot architecture - Cross-Cutting features</td>
<td>82</td>
</tr>
<tr>
<td>Clean Uncoupled Domain-Oriented Architecture</td>
<td>603</td>
</tr>
<tr>
<td>General mORMot architecture</td>
<td>71</td>
</tr>
<tr>
<td>mORMot MVC/MVVM URI - Commands sequence</td>
<td>525</td>
</tr>
<tr>
<td>Why a Client-Server ORM</td>
<td>93</td>
</tr>
<tr>
<td>Pictures</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>ORM mapping</td>
<td>92</td>
</tr>
<tr>
<td>ORM Process</td>
<td>92</td>
</tr>
<tr>
<td>Service Hosting on mORMot - shared server</td>
<td>538</td>
</tr>
<tr>
<td>Service Hosting on mORMot - two servers</td>
<td>540</td>
</tr>
<tr>
<td>Service Hosting on mORMot - one server, two instances</td>
<td>541</td>
</tr>
<tr>
<td>Service Hosting on mORMot - Content Delivery Network (CDB)</td>
<td>542</td>
</tr>
<tr>
<td>mORMot Source Code Folders</td>
<td>648</td>
</tr>
<tr>
<td>mORMot Source Code Main Units</td>
<td>102</td>
</tr>
<tr>
<td>Model View Controller process</td>
<td>86</td>
</tr>
<tr>
<td>Model View Controller concept</td>
<td>86</td>
</tr>
<tr>
<td>MVC Web and Rich Clients</td>
<td>502</td>
</tr>
<tr>
<td>SQL Aggregate via JOINed tables</td>
<td>97</td>
</tr>
<tr>
<td>NoSQL Aggregate as one document</td>
<td>97</td>
</tr>
<tr>
<td>NoSQL Graph Database</td>
<td>96</td>
</tr>
<tr>
<td>ORM Replication Classes via REST</td>
<td>182</td>
</tr>
<tr>
<td>ORM Cascaded Replication Classes via REST</td>
<td>183</td>
</tr>
<tr>
<td>ORM Real-Time Replication Classes</td>
<td>185</td>
</tr>
<tr>
<td>Corporate Servers Replication</td>
<td>185</td>
</tr>
<tr>
<td>Corporate Servers Master/Slave Replication With All Data On Main Server</td>
<td>186</td>
</tr>
<tr>
<td>Corporate Servers Master/Slave Replication With Private Local Data</td>
<td>186</td>
</tr>
<tr>
<td>Corporate Servers Master/Slave Replication With CQRS</td>
<td>187</td>
</tr>
<tr>
<td>Publish-Subscribe Pattern</td>
<td>449</td>
</tr>
<tr>
<td>Chat Application using Publish-Subscribe</td>
<td>450</td>
</tr>
<tr>
<td>Unit dependencies in the &quot;Lib\CrossPlatform&quot; directory</td>
<td>667</td>
</tr>
<tr>
<td>Unit dependencies in the &quot;Lib\SQLite3\DDD\dom&quot; directory</td>
<td>670</td>
</tr>
<tr>
<td>Unit dependencies in the &quot;Lib\SQLite3\DDD\infra&quot; directory</td>
<td>672</td>
</tr>
<tr>
<td>Unit dependencies in the &quot;Lib\SQLite3&quot; directory</td>
<td>669</td>
</tr>
<tr>
<td>Unit dependencies in the &quot;Lib\SynDBDataset&quot; directory</td>
<td>673</td>
</tr>
<tr>
<td>Unit dependencies in the &quot;Lib&quot; directory</td>
<td>666</td>
</tr>
<tr>
<td>Pictures</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Unit dependencies in the &quot;Lib\SQLite3\Samples\MainDemo&quot; directory</td>
<td>2529</td>
</tr>
<tr>
<td>Unit dependencies in the &quot;Lib\SQLite3&quot; directory</td>
<td>2528</td>
</tr>
<tr>
<td>Unit dependencies in the &quot;Lib&quot; directory</td>
<td>2527</td>
</tr>
<tr>
<td>RESTful Server classes</td>
<td>321</td>
</tr>
<tr>
<td>SOLID Principles - Single Responsibility: Single-to-rule-them-all class</td>
<td>390</td>
</tr>
<tr>
<td>SOLID Principles - Single Responsibility: Abstract parent class</td>
<td>390</td>
</tr>
<tr>
<td>SOLID Principles - Single Responsibility: Spliting protocol and communciation</td>
<td>390</td>
</tr>
<tr>
<td>RESTful storage classes</td>
<td>322</td>
</tr>
<tr>
<td>SynDB First Level Providers</td>
<td>253</td>
</tr>
<tr>
<td>SynDB and BDE</td>
<td>262</td>
</tr>
<tr>
<td>SynDB and FireDAC / AnyDAC</td>
<td>261</td>
</tr>
<tr>
<td>SynDB Architecture</td>
<td>242</td>
</tr>
<tr>
<td>SynDB and NexusDB</td>
<td>260</td>
</tr>
<tr>
<td>Oracle Connectivity with SynDBOracle</td>
<td>257</td>
</tr>
<tr>
<td>SynDB and ODBC</td>
<td>254</td>
</tr>
<tr>
<td>SynDB and OleDB</td>
<td>253</td>
</tr>
<tr>
<td>SynDB Remote access Overview</td>
<td>263</td>
</tr>
<tr>
<td>SynDB, mORMot and SQLite3</td>
<td>259</td>
</tr>
<tr>
<td>SynDB and UniDAC</td>
<td>261</td>
</tr>
<tr>
<td>SynDB and Zeos / ZDBC</td>
<td>255</td>
</tr>
<tr>
<td>TSQLRecordPeopleExt Database-First Field/Column Mapping</td>
<td>271</td>
</tr>
<tr>
<td>TSQLRecordPeopleExt Code-First Field/Column Mapping</td>
<td>270</td>
</tr>
<tr>
<td>ORM Access Via REST</td>
<td>275</td>
</tr>
<tr>
<td>ORM Access Via Virtual Table</td>
<td>276</td>
</tr>
<tr>
<td>Meet the mORMot</td>
<td>71, 76</td>
</tr>
<tr>
<td>Smart Mobile Studio Calculator Sample</td>
<td>497</td>
</tr>
<tr>
<td>User Interface generated using TMS components</td>
<td>2514</td>
</tr>
<tr>
<td>User Interface generated using VCL components</td>
<td>2515</td>
</tr>
</tbody>
</table>
Source code File Names Reference Table

The following table is a quick-reference guide to all the Source code File Names referenced in this Software Architecture Design (SAD) document.

**Others - Source Reference Table**

<table>
<thead>
<tr>
<th>Source code File Names</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lib\mORMot.pas</td>
<td>130, 420</td>
</tr>
<tr>
<td>Lib\mORMotDB.pas</td>
<td>130</td>
</tr>
<tr>
<td>Lib\mORMotSQLite3.pas</td>
<td>130</td>
</tr>
<tr>
<td>Lib\SQLite3\mORMot.pas</td>
<td>225, 641</td>
</tr>
<tr>
<td>Lib\SQLite3\mORMoti18n.pas</td>
<td>2515, 2520, 2523</td>
</tr>
<tr>
<td>Lib\SQLite3\mORMotReport.pas</td>
<td>2519</td>
</tr>
<tr>
<td>Lib\SQLite3\mORMotUIEdit.pas</td>
<td>2513</td>
</tr>
<tr>
<td>Lib\SQLite3\Samples\MainDemo\FileClient.pas</td>
<td>2506, 2519</td>
</tr>
<tr>
<td>Lib\SQLite3\Samples\MainDemo\FileEdit.pas</td>
<td>2506</td>
</tr>
<tr>
<td>Lib\SQLite3\Samples\MainDemo\FileMain.pas</td>
<td>348, 2523</td>
</tr>
<tr>
<td>Lib\SQLite3\Samples\MainDemo\FileServer.pas</td>
<td>2506</td>
</tr>
<tr>
<td>Lib\SQLite3\Samples\MainDemo\FileTables.pas</td>
<td>2506, 2507</td>
</tr>
<tr>
<td>Lib\SynCommons.pas</td>
<td>104</td>
</tr>
<tr>
<td>Lib\SynCrypto.pas</td>
<td>2506</td>
</tr>
<tr>
<td>Lib\SynDB.pas</td>
<td>130</td>
</tr>
<tr>
<td>Lib\SynGdiPlus.pas</td>
<td>2506, 2520</td>
</tr>
<tr>
<td>Lib\SynSQLite3.pas</td>
<td>109, 130</td>
</tr>
<tr>
<td>Lib\SynTests.pas</td>
<td>627</td>
</tr>
</tbody>
</table>
**Keywords Reference Table**

The following table is a quick-reference guide to all the Keywords referenced in this *Software Architecture Design* (SAD) document.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.msg</td>
<td>2523</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>53-bit</td>
<td>119, 297</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>64-bit</td>
<td>137, 213, 242, 257, 331, 646, 648</td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>ACID</td>
<td>195, 222, 222, 233</td>
</tr>
<tr>
<td>Adapter</td>
<td>600, 621, 622</td>
</tr>
<tr>
<td>AES-NI</td>
<td>334, 445, 2506</td>
</tr>
<tr>
<td>AES</td>
<td>73, 197, 212, 333, 334, 445, 571, 586</td>
</tr>
<tr>
<td>Aggregate root</td>
<td>597, 599, 601</td>
</tr>
<tr>
<td>Aggregates</td>
<td>78, 99, 597, 599, 606, 606, 608, 610</td>
</tr>
<tr>
<td>Aggregation</td>
<td>98, 158, 285</td>
</tr>
<tr>
<td>AJAX</td>
<td>72, 75, 87, 88, 156, 189, 198, 296, 314, 316, 326, 377, 384, 421, 423, 424, 468, 472, 473, 476, 553, 562, 641, 647, 2513</td>
</tr>
<tr>
<td>Android</td>
<td>73, 483, 485, 496</td>
</tr>
<tr>
<td>Anemic domain model</td>
<td>613, 616</td>
</tr>
<tr>
<td>Anti-Corruption Layer</td>
<td>600</td>
</tr>
<tr>
<td>AnyDAC</td>
<td>72, 75, 196, 241, 242, 260, 260, 356, 651</td>
</tr>
<tr>
<td>ARC</td>
<td>402</td>
</tr>
<tr>
<td>Array bind</td>
<td>77, 199, 203, 206, 255, 257, 261, 356</td>
</tr>
<tr>
<td>AsTSQLRecord</td>
<td>153</td>
</tr>
<tr>
<td>Topic</td>
<td>Page(s)</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>Asymmetric</td>
<td>571, 590</td>
</tr>
<tr>
<td>AS_UNIQUE</td>
<td>131, 135, 157, 192, 205, 271</td>
</tr>
<tr>
<td>Atomic</td>
<td>232, 544</td>
</tr>
<tr>
<td>ATTACH DATABASE</td>
<td>230, 230</td>
</tr>
<tr>
<td>Audit Trail</td>
<td>178, 463</td>
</tr>
<tr>
<td>Authentication</td>
<td>75, 130, 198, 264, 378, 380, 420, 459, 466, 544, 546</td>
</tr>
<tr>
<td>Authorization</td>
<td>459, 558</td>
</tr>
<tr>
<td>AutomaticTransactionPerRow</td>
<td>351, 494</td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>Backup</td>
<td>224</td>
</tr>
<tr>
<td>Base64</td>
<td>142, 156, 162, 296, 298, 303, 352, 423, 424, 424, 426, 469, 471, 606</td>
</tr>
<tr>
<td>BATCH</td>
<td>163, 230, 257, 261, 350, 482, 494</td>
</tr>
<tr>
<td>BDE</td>
<td>75, 77, 196, 200, 207, 241, 242, 243, 260, 261</td>
</tr>
<tr>
<td>Behavior-Driven Development</td>
<td>604</td>
</tr>
<tr>
<td>Benchmark</td>
<td>199</td>
</tr>
<tr>
<td>BigData</td>
<td>96</td>
</tr>
<tr>
<td>BinToBase64WithMagic</td>
<td>146</td>
</tr>
<tr>
<td>BLOB</td>
<td>133, 141, 146, 156, 156, 161, 198, 215, 296, 313, 352, 378, 482, 2506, 2511</td>
</tr>
<tr>
<td>BootStrap</td>
<td>519, 534</td>
</tr>
<tr>
<td>Bounded Context</td>
<td>99, 596, 597</td>
</tr>
<tr>
<td>BSON</td>
<td>112, 112, 118, 120, 279, 279, 287, 289</td>
</tr>
<tr>
<td>Business rules</td>
<td>75</td>
</tr>
<tr>
<td>By-reference</td>
<td>118</td>
</tr>
<tr>
<td>By-value</td>
<td>118</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>573</td>
</tr>
<tr>
<td>Keyword</td>
<td>Pages</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Cache</td>
<td>72, 174, 213, 317, 326, 335, 359, 518, 625</td>
</tr>
<tr>
<td>Camel</td>
<td>503, 504, 629, 2515</td>
</tr>
<tr>
<td>Cardinality</td>
<td>151, 154</td>
</tr>
<tr>
<td>CDN</td>
<td>379, 536, 542, 560</td>
</tr>
<tr>
<td>Class</td>
<td>605</td>
</tr>
<tr>
<td>Clean Architecture</td>
<td>602</td>
</tr>
<tr>
<td>Client-Server</td>
<td>71, 75, 77, 91, 94, 130, 163, 175, 189, 192, 193, 197, 198, 199, 230, 230, 313, 314, 315, 318, 319, 324, 350, 367, 393, 394, 400, 401, 439, 546, 547, 640</td>
</tr>
<tr>
<td>Cloud</td>
<td>123, 334, 481, 536, 537, 661</td>
</tr>
<tr>
<td>Code-first</td>
<td>268</td>
</tr>
<tr>
<td>Collation</td>
<td>198, 219, 227</td>
</tr>
<tr>
<td>Commands</td>
<td>600</td>
</tr>
<tr>
<td>Contract</td>
<td>372, 2547</td>
</tr>
<tr>
<td>Convention over configuration</td>
<td>72, 130, 193, 372, 478, 517</td>
</tr>
<tr>
<td>CQRS</td>
<td>77, 78, 91, 183, 187, 194, 194, 195, 610</td>
</tr>
<tr>
<td>Crc32</td>
<td>481, 553, 553, 652</td>
</tr>
<tr>
<td>Crc32c</td>
<td>379</td>
</tr>
<tr>
<td>CreateAndFillPrepare</td>
<td>144, 191</td>
</tr>
<tr>
<td>CreateJoined</td>
<td>133, 138, 153, 288</td>
</tr>
<tr>
<td>Critical Section</td>
<td>123, 335</td>
</tr>
<tr>
<td>Cross-platform</td>
<td>103, 481</td>
</tr>
<tr>
<td>CrossKylix</td>
<td>393, 630, 659</td>
</tr>
<tr>
<td>CRUD</td>
<td>81, 92, 130, 137, 144, 174, 175, 178, 193, 193, 199, 206, 232, 270, 272, 293, 317, 321, 335, 345, 351, 359, 482, 548, 550, 558, 559</td>
</tr>
<tr>
<td>CSS 3</td>
<td>80</td>
</tr>
</tbody>
</table>

**D**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAO</td>
<td>407</td>
</tr>
<tr>
<td><strong>Data Access Objects</strong></td>
<td>407</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----</td>
</tr>
<tr>
<td><strong>Data Transfer Object</strong></td>
<td>407</td>
</tr>
<tr>
<td><strong>Database-first</strong></td>
<td>268</td>
</tr>
<tr>
<td><strong>DateTimeToSQL</strong></td>
<td>146</td>
</tr>
<tr>
<td><strong>DateToSQL</strong></td>
<td>146</td>
</tr>
<tr>
<td><strong>DB2</strong></td>
<td>72, 88, 194, 195, 195, 196, 199, 199, 200, 203, 206, 207, 224, 236, 240, 242, 244, 341, 356, 357</td>
</tr>
<tr>
<td><strong>DBExpress</strong></td>
<td>72, 75, 196, 207, 241, 242</td>
</tr>
<tr>
<td><strong>DDD</strong></td>
<td>78, 400, 456</td>
</tr>
<tr>
<td><strong>Deadlock</strong></td>
<td>79, 123, 453, 453</td>
</tr>
<tr>
<td><strong>Debian</strong></td>
<td>646, 659</td>
</tr>
<tr>
<td><strong>Delphi</strong></td>
<td>380, 629</td>
</tr>
<tr>
<td><strong>Denormalization</strong></td>
<td>98, 195, 287</td>
</tr>
<tr>
<td><strong>Dependency injection</strong></td>
<td>389, 401, 406</td>
</tr>
<tr>
<td><strong>Dependency inversion</strong></td>
<td>389, 401</td>
</tr>
<tr>
<td><strong>Digital signature</strong></td>
<td>573</td>
</tr>
<tr>
<td><strong>DMZ</strong></td>
<td>538</td>
</tr>
<tr>
<td><strong>Domain-Driven</strong></td>
<td>72, 75, 81, 88, 88, 99, 99, 156, 193, 195, 433</td>
</tr>
<tr>
<td><strong>Domain</strong></td>
<td>99, 595</td>
</tr>
<tr>
<td><strong>Double</strong></td>
<td>119, 132, 137, 137, 140, 145, 214, 243, 288, 295, 297, 303, 374, 402</td>
</tr>
<tr>
<td><strong>DTO</strong></td>
<td>120, 362, 407, 600, 605, 609, 610, 620, 621</td>
</tr>
<tr>
<td><strong>Dual-phase</strong></td>
<td>601, 604, 613</td>
</tr>
<tr>
<td><strong>DvoAllowDoubleValue</strong></td>
<td>119</td>
</tr>
<tr>
<td><strong>Dynamic array</strong></td>
<td>72, 104, 107, 133, 137, 141, 141, 147, 155, 156, 156, 159, 160, 176, 189, 189, 289, 291, 296, 303, 367, 367, 517, 527, 635</td>
</tr>
</tbody>
</table>

E

E1025 | 647 |
E2201 | 647 |
ECC | 571, 576, 577 |
<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECCAuthorize</td>
<td>590</td>
</tr>
<tr>
<td>ECDSA</td>
<td>380</td>
</tr>
<tr>
<td>ECIES</td>
<td>586</td>
</tr>
<tr>
<td>Elliptic curve</td>
<td>571</td>
</tr>
<tr>
<td>Encryption</td>
<td>197, 331, 333, 334, 445, 448, 544, 571</td>
</tr>
<tr>
<td>Enhanced RTTI</td>
<td>107, 297, 304, 424, 606</td>
</tr>
<tr>
<td>Entity Objects</td>
<td>99, 596, 599, 603, 606, 606, 608</td>
</tr>
<tr>
<td>Enumerated</td>
<td>132, 288, 397, 504</td>
</tr>
<tr>
<td>Event Oriented Persistence</td>
<td>184, 622</td>
</tr>
<tr>
<td>Event Sourcing</td>
<td>195, 597, 622</td>
</tr>
<tr>
<td>Event-Driven</td>
<td>86, 444, 449, 456, 597, 621</td>
</tr>
<tr>
<td>Events</td>
<td>600, 621</td>
</tr>
<tr>
<td>Expression Helper</td>
<td>507, 514</td>
</tr>
<tr>
<td>Extended syntax</td>
<td>117</td>
</tr>
<tr>
<td>Extended syntax</td>
<td>79, 112, 113, 140, 279, 283, 284, 297, 512</td>
</tr>
</tbody>
</table>

**F**

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factory</td>
<td>99, 396, 401, 406, 425, 429, 435, 439, 596, 599, 603, 610</td>
</tr>
<tr>
<td>FAQ</td>
<td>78</td>
</tr>
<tr>
<td>FastMM4</td>
<td>106, 197, 211, 483, 486, 649, 654, 658</td>
</tr>
<tr>
<td>Filtering</td>
<td>172</td>
</tr>
<tr>
<td>Firebird</td>
<td>72, 88, 194, 195, 196, 198, 199, 200, 203, 206, 207, 224, 236, 240, 242, 244, 341, 356, 357</td>
</tr>
<tr>
<td>FireDAC</td>
<td>72, 75, 196, 200, 202, 205, 207, 208, 214, 242, 247, 260, 260, 356, 651</td>
</tr>
<tr>
<td>FireMonkey</td>
<td>483, 641</td>
</tr>
<tr>
<td>Firewall</td>
<td>325, 328</td>
</tr>
<tr>
<td>FMX</td>
<td>392</td>
</tr>
<tr>
<td>ForceBlobTransfert</td>
<td>142, 146, 2511</td>
</tr>
<tr>
<td>FPC</td>
<td>71, 79, 104, 143, 151, 192, 380, 393, 537, 630, 653, 655</td>
</tr>
<tr>
<td>Term</td>
<td>Pages</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Fpcupdeluxe</td>
<td>647, 656</td>
</tr>
<tr>
<td>FreePascal</td>
<td>73, 105, 115, 141, 254, 481, 483, 485, 495, 646, 655</td>
</tr>
<tr>
<td>FTS</td>
<td>197, 215, 215, 216, 217, 225</td>
</tr>
<tr>
<td>Full Text</td>
<td>158, 197, 521</td>
</tr>
<tr>
<td>Garbage collector</td>
<td>150, 403, 432, 567</td>
</tr>
<tr>
<td>Gateway</td>
<td>422, 479</td>
</tr>
<tr>
<td>General Public License</td>
<td>60, 643</td>
</tr>
<tr>
<td>GitHub</td>
<td>80, 646, 648, 653</td>
</tr>
<tr>
<td>GridFS</td>
<td>289</td>
</tr>
<tr>
<td>GUID</td>
<td>141, 303, 311, 333, 386, 422, 434, 461</td>
</tr>
<tr>
<td>Has many through</td>
<td>154, 165</td>
</tr>
<tr>
<td>Has many</td>
<td>154</td>
</tr>
<tr>
<td>Hexagonal architecture</td>
<td>600, 602</td>
</tr>
<tr>
<td>HMAC-SHA256</td>
<td>381, 544, 576, 586</td>
</tr>
<tr>
<td>HMAC-SHA2</td>
<td>380</td>
</tr>
<tr>
<td>HMAC</td>
<td>586</td>
</tr>
<tr>
<td>Hosting</td>
<td>73, 429, 537, 624</td>
</tr>
<tr>
<td>HTML 5</td>
<td>80</td>
</tr>
<tr>
<td>Http.sys</td>
<td>75, 325, 326, 479, 652, 654</td>
</tr>
<tr>
<td>HTTPS</td>
<td>72, 75, 325, 326, 331, 546, 546, 553, 571</td>
</tr>
<tr>
<td>HTTP_RESP_STATICFILE</td>
<td>327</td>
</tr>
<tr>
<td>I18n</td>
<td>72, 105, 135, 504, 504, 515, 515, 651, 2515, 2520, 2523</td>
</tr>
<tr>
<td>Index</td>
<td>135, 141, 161, 192, 268, 291, 520</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Indy</td>
<td>485</td>
</tr>
<tr>
<td>Informix</td>
<td>72, 88, 194, 195, 196, 199, 207, 224, 236, 240, 244, 341, 356, 357</td>
</tr>
<tr>
<td>IntegerDynArrayContains</td>
<td>156, 161, 367</td>
</tr>
<tr>
<td>Interface</td>
<td>75, 91, 385, 389, 399, 399, 400, 401, 439</td>
</tr>
<tr>
<td>Inversion Of Control</td>
<td>389, 401, 406, 417</td>
</tr>
<tr>
<td>IoC</td>
<td>389, 401, 406, 417, 417, 529, 614, 617, 618, 618</td>
</tr>
<tr>
<td>IOS</td>
<td>485</td>
</tr>
<tr>
<td>Ipad</td>
<td>73, 483, 496</td>
</tr>
<tr>
<td>Iphone</td>
<td>73, 483, 496</td>
</tr>
<tr>
<td>ISO 8601</td>
<td>120, 132, 132, 134, 136, 197, 198, 288, 303, 423, 482</td>
</tr>
<tr>
<td>ISQLDBRows</td>
<td>247</td>
</tr>
</tbody>
</table>

J

| JavaScript | 75, 80, 88, 91, 119, 188, 296, 297, 316, 486, 505, 546, 556, 562, 647, 648 |
| Jet/MSAccess | 72, 88, 194, 195, 196, 199, 200, 207, 240, 244, 337 |
| JOIN | 147, 154, 169, 197, 206, 225, 236, 345 |
| JSON-RPC | 421, 466, 478 |
| JWT | 379, 531, 557 |

K

<p>| Kerberos | 554, 558 |
| KISS | 76, 104, 193, 335, 372, 479, 566 |</p>
<table>
<thead>
<tr>
<th><strong>L10n</strong></th>
<th>2523</th>
</tr>
</thead>
<tbody>
<tr>
<td>Late-binding</td>
<td>112, 113, 113, 116, 116, 120, 147, 157, 243, 249, 257, 562, 564, 568, 569, 570, 620</td>
</tr>
<tr>
<td>Layer Supertype</td>
<td>193, 600, 606</td>
</tr>
<tr>
<td>Lazarus</td>
<td>73, 254, 392, 486, 495, 646</td>
</tr>
<tr>
<td>Lazy loading</td>
<td>154, 155, 165</td>
</tr>
<tr>
<td>Lesser General Public License</td>
<td>643</td>
</tr>
<tr>
<td>Libcurl</td>
<td>329, 661</td>
</tr>
<tr>
<td>License</td>
<td>393, 643</td>
</tr>
<tr>
<td>Linux</td>
<td>71, 200, 206, 264, 393, 481, 487, 496, 537, 577, 646, 647, 648, 659</td>
</tr>
<tr>
<td>Liskov substitution principle</td>
<td>395</td>
</tr>
<tr>
<td>Log</td>
<td>104, 248, 621, 631, 634, 635, 2513</td>
</tr>
</tbody>
</table>

## M

<table>
<thead>
<tr>
<th>Many to many</th>
<th>138</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map/reduce</td>
<td>98, 113, 158, 285, 292, 358, 464</td>
</tr>
<tr>
<td>MapAutoKeywordFields</td>
<td>273</td>
</tr>
<tr>
<td>Mapping</td>
<td>271</td>
</tr>
<tr>
<td>Markdown</td>
<td>515</td>
</tr>
<tr>
<td>Master/Detail</td>
<td>151, 155, 156, 165, 176, 189, 189</td>
</tr>
<tr>
<td>Master/slave</td>
<td>180, 185</td>
</tr>
<tr>
<td>Message bus</td>
<td>621</td>
</tr>
<tr>
<td>MicroService</td>
<td>78</td>
</tr>
<tr>
<td>MIME</td>
<td>374, 377</td>
</tr>
<tr>
<td>Mock</td>
<td>72, 399, 407, 408, 410, 518</td>
</tr>
<tr>
<td>Model</td>
<td>86, 99, 178, 344, 502, 548, 596, 598</td>
</tr>
<tr>
<td>Mozilla Public License</td>
<td>643</td>
</tr>
<tr>
<td>Term</td>
<td>Pages</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>MS SQL</td>
<td>72, 88, 158, 194, 195, 196, 199, 200, 206, 207, 209, 224, 236, 240, 242, 244, 247, 253, 268, 270, 277, 277, 336, 341, 356, 357, 366, 441, 551</td>
</tr>
<tr>
<td>Multi-thread</td>
<td>123, 277, 335, 453, 461, 621</td>
</tr>
<tr>
<td>Mustache</td>
<td>72, 72, 77, 80, 87, 489, 502, 505, 516, 522, 533, 650</td>
</tr>
<tr>
<td>MVC</td>
<td>71, 72, 78, 80, 86, 103, 171, 502, 502, 504, 505, 516, 516, 519, 522, 652, 2510</td>
</tr>
<tr>
<td>MVVM</td>
<td>80, 462, 517, 519</td>
</tr>
<tr>
<td>MySQL</td>
<td>72, 88, 156, 194, 195, 196, 199, 199, 200, 203, 203, 206, 207, 209, 224, 236, 236, 240, 242, 244, 255, 261, 341, 356, 357</td>
</tr>
<tr>
<td>N-Tier</td>
<td>277, 341, 623</td>
</tr>
<tr>
<td>NexusDB</td>
<td>72, 75, 88, 194, 195, 196, 199, 200, 203, 206, 207, 207, 240, 241, 242, 244, 260, 260, 357, 651</td>
</tr>
<tr>
<td>Node.js</td>
<td>565</td>
</tr>
<tr>
<td>Normalization</td>
<td>98, 195</td>
</tr>
<tr>
<td>NoSQL</td>
<td>72, 75, 77, 88, 96, 130, 140, 144, 156, 157, 159, 179, 195, 234, 239, 278, 279, 287, 322, 354, 357, 366, 397, 537, 559, 602, 616, 640, 650, 652</td>
</tr>
<tr>
<td>NTLM</td>
<td>554, 558</td>
</tr>
<tr>
<td>NULL</td>
<td>142</td>
</tr>
<tr>
<td>O-DBC</td>
<td>72, 75, 196, 198, 200, 207, 241, 242, 245, 254, 357, 650</td>
</tr>
<tr>
<td>ODM</td>
<td>98, 120, 194, 279, 287, 652</td>
</tr>
<tr>
<td>OIC</td>
<td>257, 258</td>
</tr>
<tr>
<td>OleDB</td>
<td>72, 75, 196, 198, 207, 241, 242, 245, 253, 357, 441, 650</td>
</tr>
<tr>
<td>One to many</td>
<td>138, 151, 151</td>
</tr>
<tr>
<td>One to one</td>
<td>138, 151, 151</td>
</tr>
<tr>
<td>OOP</td>
<td>80, 98, 177, 178, 242, 251, 266, 385, 389, 393, 487, 599, 605, 2507</td>
</tr>
<tr>
<td>Oracle Instant Client</td>
<td>257</td>
</tr>
<tr>
<td>Oracle Wallet</td>
<td>258</td>
</tr>
<tr>
<td>---------------</td>
<td>-----</td>
</tr>
<tr>
<td>OSX</td>
<td>73, 481, 483</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packages</td>
</tr>
<tr>
<td>PBKDF2_HMAC_SHA256</td>
</tr>
<tr>
<td>Pdf</td>
</tr>
<tr>
<td>Perfect forward secrecy</td>
</tr>
<tr>
<td>Persistence Ignorance</td>
</tr>
<tr>
<td>PhoneGap</td>
</tr>
<tr>
<td>PKI</td>
</tr>
<tr>
<td>PODO</td>
</tr>
<tr>
<td>Prepared</td>
</tr>
<tr>
<td>Primary key</td>
</tr>
<tr>
<td>Proxy</td>
</tr>
<tr>
<td>Public-key</td>
</tr>
<tr>
<td>Publish-subscribe</td>
</tr>
<tr>
<td>Publish/subscribe</td>
</tr>
<tr>
<td>Published method</td>
</tr>
<tr>
<td>Published properties</td>
</tr>
<tr>
<td>Q</td>
</tr>
<tr>
<td>R</td>
</tr>
</tbody>
</table>

| Race condition | 123 |
| RAD | 75, 79, 272 |
| RawByteString | 105, 133, 244, 289, 313, 423, 471 |
| RawJSON | 423, 473 |
| RawUTF8 | 105, 110, 132, 135, 145, 147, 162, 164, 168, 191, 214, 217, 243, 288, 297, 303, 304, 423, 483 |
| RDBMS | 92, 151, 156, 241 |
| Record | 297, 298, 424, 426, 605 |
| RecordLoad | 298 |
| RecordRef | 133, 139, 289, 2509, 2517 |
| RecordSave | 298 |
| Redirection | 72, 185, 233, 538 |
| Reference-counted | 387 |
| REGEXP | 221 |
| RegisterClassForJSON | 133, 156, 289 |
| RegisterCustomJSONSerializer | 299, 304 |
| RegisterCustomJSONSerializerFromText | 300, 304 |
| RegisterCustomSerializer | 133, 156, 165, 289, 307, 635 |
| Regular expression | 198, 221 |
| Replication | 72, 96, 180, 180, 185, 189, 236 |
| Report | 75, 2506 |
| Repository | 99, 396, 407, 425, 596, 600, 603, 610 |
| ResourceString | 2521, 2523 |
### Responsive design


### REST

71, 72, 107, 107, 131, 137, 141, 148, 193, 503, 635, 2506, 2513, 2515, 2516, 2523

### RESTful

158, 197, 214, 225

### RTREE

248, 333, 481, 547, 550, 553, 558, 621, 641

### RTTI

72, 75, 197, 419, 459, 473, 504, 538, 544, 621, 641

### SAX

105, 159

### Schema-less

157

### Script

561

### Seam

77, 100

### Security

72, 75, 197, 419, 459, 473, 504, 538, 544, 621, 641

### Serialization

75, 91, 107, 109, 162, 296, 296, 296

### Server time stamp

177

### Service

81, 87, 90, 99, 162, 176, 192, 311, 321, 365, 372, 601, 603, 603, 620, 624, 2506, 2547

### ServiceContext

337, 432, 433

### Session

72, 163, 361, 378, 430, 544, 546, 547, 547, 547, 550, 553, 558, 621, 641

### Sessions

530

### SHA256

333, 481, 547, 549, 551, 553, 553, 571, 584, 586, 652

### SHA3

380

### Sharding

154, 155

### Shared nothing architecture

155

### Singleton

394

### Smart Mobile Studio

73, 80, 470, 481, 485, 486, 487, 489, 496, 562, 641, 654

### SmartPascal

486, 488, 498, 500, 557
<table>
<thead>
<tr>
<th>SynDBExplorer</th>
<th>SynDB</th>
<th>Synchronize</th>
<th>Stub</th>
<th>Strong type</th>
<th>Stored procedure</th>
<th>StaticMongoDBRegister</th>
<th>StaticMongoDBRegisterAll</th>
<th>Static</th>
<th>Stateless</th>
<th>Stand-alone</th>
<th>Stateless</th>
<th>Static</th>
<th>StaticMongoDBRegister</th>
<th>StaticMongoDBRegisterAll</th>
<th>Stored procedure</th>
<th>Strong type</th>
<th>Stub</th>
<th>Synchronize</th>
<th>SynDB</th>
<th>SynDBExplorer</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynDprUses.inc</td>
<td>658</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SynFastWideString.pas</td>
<td>106, 2522</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SynLZ</td>
<td>226, 297, 326, 333, 353, 358, 638, 650</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synopse.inc</td>
<td>104</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SynProject</td>
<td>624, 646, 649, 650, 2510</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SynUnicode</td>
<td>105, 132, 288, 304, 423</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### T

<table>
<thead>
<tr>
<th>TAutoLocker</th>
<th>462, 530</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBSONVariant</td>
<td>112, 279</td>
</tr>
<tr>
<td>TClientDataSet</td>
<td>243, 245, 250</td>
</tr>
<tr>
<td>TCollection</td>
<td>107, 133, 155, 156, 156, 162, 189, 189, 289, 422, 423, 635</td>
</tr>
<tr>
<td>TCreateTime</td>
<td>122, 133, 136, 146, 177, 197, 288, 482</td>
</tr>
<tr>
<td>TDataSet</td>
<td>207, 241, 243, 245, 250, 250, 254, 263, 651</td>
</tr>
<tr>
<td>TDateTime</td>
<td>120, 132, 136, 146, 288, 423, 507, 514, 515</td>
</tr>
<tr>
<td>TDateTimeMS</td>
<td>120, 121, 132, 136, 288, 423</td>
</tr>
<tr>
<td>TDD</td>
<td>401</td>
</tr>
<tr>
<td>TDecimal128</td>
<td>119</td>
</tr>
<tr>
<td>TDocVariant</td>
<td>104, 112, 112, 128, 158, 464, 517, 593</td>
</tr>
<tr>
<td>TDynArray</td>
<td>111, 111, 111, 304, 304</td>
</tr>
<tr>
<td>TDynArrayHashed</td>
<td>111, 111, 243</td>
</tr>
<tr>
<td>Template</td>
<td>502, 505</td>
</tr>
<tr>
<td>Test-Driven</td>
<td>72, 76, 401</td>
</tr>
<tr>
<td>Test</td>
<td>72, 81, 104, 105, 159, 162, 162, 166, 217, 351, 399, 627, 639, 642, 650, 652, 654</td>
</tr>
<tr>
<td>Text Search</td>
<td>75</td>
</tr>
<tr>
<td>TGUID</td>
<td>141</td>
</tr>
<tr>
<td>Thread-safe</td>
<td>198, 335, 558</td>
</tr>
<tr>
<td>TID</td>
<td>131, 133, 138, 157, 214, 289, 662</td>
</tr>
<tr>
<td>Tier</td>
<td>72, 72, 74, 75, 77, 81, 88</td>
</tr>
<tr>
<td>Class</td>
<td>Page Numbers</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Time zone</td>
<td>122</td>
</tr>
<tr>
<td>TimeLogToSQL</td>
<td>146</td>
</tr>
<tr>
<td>TInjectableObject</td>
<td>127</td>
</tr>
<tr>
<td>TInjectableObjectRest</td>
<td>433, 434</td>
</tr>
<tr>
<td>TInterfacedCollection</td>
<td>422, 423, 423</td>
</tr>
<tr>
<td>TLockedDocVariant</td>
<td>462, 530</td>
</tr>
<tr>
<td>TModTime</td>
<td>122, 132, 136, 137, 146, 177, 197, 288, 482</td>
</tr>
<tr>
<td>TMonitor</td>
<td>124</td>
</tr>
<tr>
<td>TMS</td>
<td>646, 2513</td>
</tr>
<tr>
<td>TNullable</td>
<td>142</td>
</tr>
<tr>
<td>TNullableBoolean</td>
<td>133</td>
</tr>
<tr>
<td>TNullableCurrency</td>
<td>134</td>
</tr>
<tr>
<td>TNullableDateTime</td>
<td>134</td>
</tr>
<tr>
<td>TNullableFloat</td>
<td>134</td>
</tr>
<tr>
<td>TNullableInteger</td>
<td>133</td>
</tr>
<tr>
<td>TNullableTimeLog</td>
<td>134</td>
</tr>
<tr>
<td>TNullableUTF8Text</td>
<td>134</td>
</tr>
<tr>
<td>TObject</td>
<td>133, 156, 289, 306, 423</td>
</tr>
<tr>
<td>TObjectList</td>
<td>133, 156, 156, 165, 289, 309, 423</td>
</tr>
<tr>
<td>TPersistent</td>
<td>133, 155, 156, 162, 189, 189, 289, 306, 376, 423, 446, 635, 2507</td>
</tr>
<tr>
<td>TQuery</td>
<td>77, 243, 250, 263, 651</td>
</tr>
<tr>
<td>Transaction</td>
<td>163, 166, 167, 232, 313, 315, 351, 550</td>
</tr>
<tr>
<td>TRawUTF8List</td>
<td>133, 162, 289, 306</td>
</tr>
<tr>
<td>TRecordReference</td>
<td>133, 139, 189, 234, 289, 662, 2509</td>
</tr>
<tr>
<td>TRecordReferenceToBeDeleted</td>
<td>133, 139, 289</td>
</tr>
<tr>
<td>TRecordVersion</td>
<td>134, 180</td>
</tr>
<tr>
<td>TServiceCustomAnswer</td>
<td>423, 424, 476</td>
</tr>
<tr>
<td>TServiceFactory</td>
<td>406, 433</td>
</tr>
<tr>
<td>TSessionUserID</td>
<td>133, 136</td>
</tr>
<tr>
<td>Class</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TSMEngine</td>
<td>566</td>
</tr>
<tr>
<td>TSMEngineManager</td>
<td>566</td>
</tr>
<tr>
<td>TSMVariant</td>
<td>562, 564, 569</td>
</tr>
<tr>
<td>TSQLAuthGroup</td>
<td>442, 548</td>
</tr>
<tr>
<td>TSQLAuthUser</td>
<td>378, 442, 548</td>
</tr>
<tr>
<td>TSQLDataBase</td>
<td>317</td>
</tr>
<tr>
<td>TSQLModel</td>
<td>86, 131, 171, 173, 189, 190, 230, 231, 327, 328, 502, 2506, 2509</td>
</tr>
<tr>
<td>TSQLModelRecordProperties</td>
<td>171, 273</td>
</tr>
<tr>
<td>TSQLRawBlob</td>
<td>133, 141, 142, 146, 146, 289, 289, 313</td>
</tr>
<tr>
<td>TSQLRecordMany</td>
<td>133, 155, 155, 156, 162, 165, 169, 189, 289</td>
</tr>
<tr>
<td>TSQLRecordMappedAutoID</td>
<td>194</td>
</tr>
<tr>
<td>TSQLRecordMappedForcedID</td>
<td>194</td>
</tr>
<tr>
<td>TSQLRecordProperties</td>
<td>171</td>
</tr>
<tr>
<td>TSQLRecordTableDelete</td>
<td>180</td>
</tr>
<tr>
<td>TSQLRecordVirtual</td>
<td>194</td>
</tr>
<tr>
<td>TSQLRecordVirtualTableAutoID</td>
<td>229, 231, 237</td>
</tr>
<tr>
<td>TSQLRecordVirtualTableForcedID</td>
<td>230</td>
</tr>
<tr>
<td>TSQLRest</td>
<td>139, 147, 171, 189, 213, 313, 314, 321, 324, 2509</td>
</tr>
<tr>
<td>TSQLRestBatch</td>
<td>353</td>
</tr>
<tr>
<td>TSQLRestClient</td>
<td>324</td>
</tr>
<tr>
<td>TSQLRestClientDB</td>
<td>231, 324, 343, 558</td>
</tr>
<tr>
<td>TSQLRestClientRedirect</td>
<td>322, 324</td>
</tr>
<tr>
<td>TSQLRestServer</td>
<td>78, 319, 324</td>
</tr>
<tr>
<td>Class</td>
<td>Pages</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>TSQLRestServerDB</td>
<td>78, 110, 198, 199, 215, 222, 223, 224, 231, 259, 266, 317, 321, 324, 335, 343, 369, 371, 373, 429, 438</td>
</tr>
<tr>
<td>TSQLRestServerFullMemory</td>
<td>78, 198, 321, 373, 429, 438</td>
</tr>
<tr>
<td>TSQLRestServerRemoteDB</td>
<td>322, 540</td>
</tr>
<tr>
<td>TSQLRestServerURIContext</td>
<td>662</td>
</tr>
<tr>
<td>TSQLRestStorage</td>
<td>322</td>
</tr>
<tr>
<td>TSQLRestStorageInMemory</td>
<td>171, 194, 195, 226, 227, 230, 231, 232, 279, 335</td>
</tr>
<tr>
<td>TSQLTableJSON</td>
<td>145, 146, 148, 347</td>
</tr>
<tr>
<td>TSQLVirtualTableBinary</td>
<td>225, 226, 230</td>
</tr>
<tr>
<td>TSQLVirtualTableJSON</td>
<td>225, 226, 230</td>
</tr>
<tr>
<td>TStrings</td>
<td>133, 156, 156, 189, 189, 289, 306</td>
</tr>
<tr>
<td>TSynDictionary</td>
<td>111, 129, 362</td>
</tr>
<tr>
<td>TSynLocker</td>
<td>124</td>
</tr>
<tr>
<td>TSynSQLStatementDataSet</td>
<td>243, 245, 250</td>
</tr>
<tr>
<td>TSynTimeZone</td>
<td>123</td>
</tr>
<tr>
<td>TTimeLog</td>
<td>121, 132, 134, 136, 146, 197, 288, 303, 507, 514, 515, 2520</td>
</tr>
<tr>
<td>TTimeLogBits</td>
<td>121, 136, 662</td>
</tr>
<tr>
<td>TNulllableUTF8Text</td>
<td>135</td>
</tr>
<tr>
<td>TUunixMSTime</td>
<td>121, 122, 133, 136, 146, 197, 288</td>
</tr>
<tr>
<td>TUunixTime</td>
<td>122, 133, 136, 146, 197, 288, 303, 2520</td>
</tr>
</tbody>
</table>

**U**

<table>
<thead>
<tr>
<th>Class</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubiquitous Language</td>
<td>99, 456, 598</td>
</tr>
<tr>
<td>Ubuntu</td>
<td>646, 659</td>
</tr>
<tr>
<td>Unicode</td>
<td>220</td>
</tr>
<tr>
<td>UniDAC</td>
<td>72, 75, 196, 200, 207, 208, 241, 242, 260, 261, 357, 651</td>
</tr>
<tr>
<td>Unit Of Work</td>
<td>353, 354, 596, 601, 614</td>
</tr>
<tr>
<td>UseHttpApi</td>
<td>328</td>
</tr>
<tr>
<td>User-Agent</td>
<td>79, 378</td>
</tr>
<tr>
<td>UTC</td>
<td>122</td>
</tr>
<tr>
<td>V</td>
<td>Validation</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td>Value Objects</td>
</tr>
<tr>
<td></td>
<td>Variant</td>
</tr>
<tr>
<td></td>
<td>Virtual Table</td>
</tr>
<tr>
<td></td>
<td>VirtualTableExternalMap</td>
</tr>
<tr>
<td></td>
<td>VirtualTableExternalRegister</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W</th>
<th>WCF</th>
<th>72, 372, 478</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Weak pointers</td>
<td>402</td>
</tr>
<tr>
<td></td>
<td>Web Application</td>
<td>71, 80, 103, 349, 496, 508, 515, 518, 519, 522, 530, 542, 652</td>
</tr>
<tr>
<td></td>
<td>WebSockets</td>
<td>180, 325, 330, 334, 419, 420, 420, 444, 444, 450, 453, 621, 649</td>
</tr>
<tr>
<td></td>
<td>WideString</td>
<td>105, 106, 132, 243, 288, 303, 402, 423</td>
</tr>
<tr>
<td></td>
<td>Wiki</td>
<td>515</td>
</tr>
<tr>
<td></td>
<td>Win64</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td>Windows Authentication</td>
<td>551, 553, 556</td>
</tr>
<tr>
<td></td>
<td>Windows</td>
<td>537, 577</td>
</tr>
</tbody>
</table>

| X       | XML         | 474 |

<table>
<thead>
<tr>
<th>Z</th>
<th>ZDBC</th>
<th>194, 198, 200, 207, 207, 242, 245, 254, 357</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Zeos</td>
<td>72, 75, 196, 199, 200, 207, 241, 254, 651</td>
</tr>
<tr>
<td></td>
<td>Zeroing Weak pointers</td>
<td>402</td>
</tr>
</tbody>
</table>
Foreword

The whole Software documentation process follows the typical steps of this diagram:

Purpose

This Software Architecture Design (SAD) document applies to the 1.18 release of the Synopse mORMot Framework library.

After a deep presentation of the framework architecture and main features, each source code unit is detailed, with clear diagrams and tables showing the dependencies between the units, and the class hierarchy of the objects implemented within.

The SynFile main demo is presented on its own, and can be used as a general User Guide of its basic ORM features and User Interface generation - see below (page 2505).
At the end of this document, *Software Requirements Specifications* (SWRS) document items are linked directly to the class or function involved with the *Software Design Document* (SDD) document, from the source code.

**Responsibilities**

- Support is available in the project forum - [https://synopse.info/forum..](https://synopse.info/forum..) - from the mORMot Open Source community;
- Tickets can be created in a public Tracker web site located at [https://synopse.info/fossil..](https://synopse.info/fossil..), which publishes also the latest version of the project source code;
- Synopse can provide additional support, expertise or enhancements, on request;
- Synopse work on the framework is distributed without any warranty, according to the chosen license terms - see below (page 643);
- This documentation is released under the GPL (*GNU General Public License*) terms, without any warranty of any kind.

**GNU General Public License**

---

**GNU GENERAL PUBLIC LICENSE**

Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License.
giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.


You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive
interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.
A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.
Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this license with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is...
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License. If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same
material under section 10.


You are not required to accept this License in order to receive or
run a copy of the Program. Ancillary propagation of a covered work
occurring solely as a consequence of using peer-to-peer transmission
to receive a copy likewise does not require acceptance. However,
nothing other than this License grants you permission to propagate or
modify any covered work. These actions infringe copyright if you do
not accept this License. Therefore, by modifying or propagating a
covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically
receives a license from the original licensors, to run, modify and
propagate that work, subject to this License. You are not responsible
for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations. If propagation of a covered
work results from an entity transaction, each party to that
transaction who receives a copy of the work also receives whatever
licenses to the work the party's predecessor in interest had or could
give under the previous paragraph, plus a right to possession of the
Corresponding Source of the work from the predecessor in interest, if
the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the
rights granted or affirmed under this License. For example, you may
not impose a license fee, royalty, or other charge for exercise of
rights granted under this License, and you may not initiate litigation
(including a cross-claim or counterclaim in a lawsuit) alleging that
any patent claim is infringed by making, using, selling, offering for
sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this
License of the Program or a work on which the Program is based. The
work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims
owned or controlled by the contributor, whether already acquired or
hereafter acquired, that would be infringed by some manner, permitted
by this License, of making, using, or selling its contributor version,
but do not include claims that would be infringed only as a
consequence of further modification of the contributor version. For
purposes of this definition, "control" includes the right to grant
patent sublicenses in a manner consistent with the requirements of
this License.
Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.


If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this license would be to refrain entirely from conveying the Program.
13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided...
above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program.  If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year>  <name of author>
This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.
1. Synopse mORMot Overview

**Meet the mORMot**

*Synopse mORMot* is an Open Source Client-Server ORM SOA MVC framework for *Delphi* 6 up to *Delphi 10.3 Rio* and FPC, targeting *Win/Linux* for the server, and any platform for clients (including mobile or AJAX).

The main features of *mORMot* are therefore:
- **ORM/ODM**: objects persistence on almost any database (SQL or NoSQL);
- **SOA**: organize your business logic into REST services;
- **Clients**: consume your data or services from any platform, via ORM classes or SOA interfaces;
- **Web MVC**: publish your ORM/SOA process as responsive Web Applications.

With local or remote access, via an auto-configuring Client-Server REST design.
**General mORMot architecture**

*mORMot* offers all features needed for building any kind of modern software project, with state-of-the-art integrated software components, designed for both completeness and complementarity, offering **convention over configuration** solutions, and implemented for speed and efficiency.

*For storing some data*, you define a class, and the framework will take care of everything: routing, JSON marshalling, table creation, SQL generation, validation.

*For creating a service*, you define an interface and a class, and you are done. Of course, the same ORM/ODM or SOA methods will run on both server and client sides: code once, use everywhere!

*For building a MVC web site*, write a Controller class in Delphi, then some HTML Views using *Mustache* templates, leveraging the same ORM/ODM or SOA methods as Model.

If you need a HTTP server, a proxy redirection, master/slave replication, publish-subscribe, a test, a mock, add security, define users or manage rights, a script engine, a report, User Interface, switch to XML format or publish HTML dynamic pages - just pick up the right class or method. If you need a tool or feature, it is probably already there, waiting for you to use it.

The table content of this document makes it clear: this is no ordinary piece of software.

The *mORMot* framework provides an Open Source self-sufficient set of units (even *Delphi* starter edition is enough) for creating any Multi-tier application, up to the most complex Domain-Driven design - see below (page 99):

- **Presentation layer** featuring MVC UI generation with i18n and reporting for rich *Delphi* clients, *Mustache*-based templates for web views - see below (page 519) - or rich AJAX clients;
- **Application layer** implementing Service Oriented Architecture via interface-based services (like *WCF*) and Client-Server ORM - following a RESTful model using JSON over several communication protocols (e.g. HTTP/1.1 and HTTPS);
- **Domain Model layer** handling all the needed business logic in plain *Delphi* objects, including high-level managed types like dynamic arrays or records for Value Objects, dedicated classes for Entities or Aggregates, and variant storage with late-binding for dynamic documents - your business logic may also be completed in *JavaScript* on the server side as stated below (page 561);
- **Data persistence infrastructure layer** with ORM persistence on direct Oracle, MS SQL, OLEDB, ODBC, Zeos connection or any DB.pas provider (e.g. NexusDB, DBExpress, FireDAC, AnyDAC, UniDAC...), with a powerful SQLite3 kernel, and direct SQL access if needed - including SQL auto-generation for SQLite3, Oracle, Jet/MSAccess, MS SQL, Firebird, DB2, PostgreSQL, MySQL, Informix and NexusDB - the ORM is also able to use NoSQL engines via a native MongoDB connection, for ODM persistence;
- **Cross-Cutting infrastructure layers** for handling data filtering and validation, security, session, cache, logging and testing (framework uses test-driven approach and features stubbing and mocking).

If you do not know some of those concepts, don't worry: this document will detail them - see below (page 81).

With *mORMot*, ORM is not used only for data persistence of objects in databases (like in other implementations), but as part of a global n-Tier, Service Oriented Architecture (SOA), ready to implement Domain-Driven solutions.

*mORMot* is not another ORM on which a transmission layer has been added, like almost everything existing in Delphi, C# or Java: this is a full Client-Server ORM/SOA from the ground up. This really makes the difference.

The business logic of your applications will be easily exposed as Services, and will be accessible from
light clients (written in Delphi or any other mean, including AJAX).

The framework Core is non-visual: it provides only a set of classes to be used from code. But you have also some UI units available (including screen auto-creation, reporting and ribbon GUI), and you can use it from any RAD, web, or AJAX clients.

No dependency is needed at the client side (no DB driver, or third-party runtime): it is able to connect via standard HTTP or HTTPS, even through a corporate proxy or a VPN. Rich Delphi clients can be deployed just by copying and running a stand-alone small executable, with no installation process. Client authentication is performed via several secure methods, and communication can be encrypted via HTTS or with a proprietary SHA/AES-256 algorithm. SOA endpoints are configured automatically for each published interface on both server and client sides, and creating a load-balancing proxy is a matter of one method call. Changing from one database engine to another is just a matter of one line of code; full audit-trail history is available, if needed, to track all changes of any class persisted by the ORM/ODM.

Cross-platform clients can be easily created, as Win32 and Win64 executables of course, but also for any platform supported by the Delphi compiler (including Mac OSX, iPhone/iPad and Android), or by FreePascal / Lazarus. AJAX applications can easily be created via Smart Mobile Studio, as will any mobile operating system be accessible as an HTML5 web rich client or stand-alone PhoneGap application, ready to be added to the Windows, Apple or Google store. See below (page 481) for how mORMot client code generation leverages all platforms.

Speed and scalability has been implemented from the ground up - see below (page 199): a genuine optimized multi-threaded core let a single server handle more than 50,000 concurrent clients, faster than DataSnap, WCF or node.js, and our rich SOA design is able to implement both vertical and horizontal scalable hosting, using recognized enterprise-level SQL or NoSQL databases for storage.

In short, with mORMot, your ROI is maximized.
1.1. Client-Server ORM/SOA framework

The Synopse mORMot framework implements a Client-Server RESTful architecture, trying to follow some MVC, N-Tier, ORM, SOA best-practice patterns - see below (page 81).

Several clients, can access to the same remote or local server, using diverse communication protocols:

General mORMot architecture - Client / Server

Or the application can be stand-alone:

General mORMot architecture - Stand-alone application

Switch from this embedded architecture to the Client-Server one is just a matter of how mORMot classes are initialized. For instance, the very same executable can even be running as a stand-alone application, a server, or a client, depending on some run-time parameters!
1.2. Highlights

At first, some points can be highlighted, which make this framework distinct to other available solutions:

- Client-Server orientation, with optimized request caching and intelligent update over a RESTful architecture - but can be used in stand-alone applications;
- No RAD components, but true ORM and SOA approach;
- Multi-Tier architecture, with integrated Business rules as fast ORM-based classes and Domain-Driven design;
- Service-Oriented-Architecture model, using custom RESTful JSON services - you can send as JSON any TStrings, TCollection, TPersistent or TObject (via registration of a custom serializer) instance, or even a *dynamic array*, or any record content, with integrated JSON serialization, via an interface-based contract shared on both client and server sides;
- Truly RESTful authentication with a dual security model (session + per-query);
- Very fast JSON producer and parser, with caching at SQL level;
- Fast a configuration-less HTTP / HTTPS server using http.sys kernel-mode server - but may communicate via named pipes, Windows Messages or in-process as lighter alternatives;
- Using *SQLite3* as its kernel, but able to connect to any other database (via OleDB / ODBC / Zeos or direct client library access e.g. for Oracle) - the SynDB.pas classes are self-sufficient, and do not depend on the *Delphi* DB.pas unit nor any third-party (so even the *Delphi* Starter edition is enough) - but the SynDBDataset unit is also available to access any DB.pas based solution (e.g. NexusDB, DBExpress, FireDAC, AnyDAC, UniDAC or even the BDE...);
- RESTful ORM access to a NoSQL database engine like *MongoDB* with the same code base;
- Ability to use SQL and RESTful requests over multiple databases at once (thanks to *SQLite3* unique Virtual Tables mechanism);
- Full Text Search engine included, with enhanced Google-like ranking algorithm;
- Server-side JavaScript engine, for defining your business intelligence;
- Direct User Interface generation: grids are created on the fly, together with a modern Ribbon (‘Office 2007’-like) screen layout - the code just has to define actions, and assign them to the tables, in order to construct the whole interface from a few lines of code, without any IDE usage;
- Integrated Reporting system, which could serve complex PDF reports from your application;
- Designed to be as fast as possible (asm used when needed, buffered reading and writing avoid most memory consumption, multi-thread ready architecture...) so benchmarks sound impressive when compared to other solutions - see below (page 199);
- More than 1800 pages of documentation;
- *Delphi*, *FreePascal*, mobile and AJAX clients can share the same server, and ORM/SOA client access code can be generated on request for any kind of application - see below (page 481);
- Full source code provided - so you can enhance it to fulfill any need;
- Works from *Delphi 6* up to *Delphi 10.3 Rio* and FPC 2.6.4/2.7.1/3.x, truly Unicode (uses UTF-8 encoding in its kernel, just like JSON), with any version of *Delphi* (no need to upgrade your IDE).
1.3. Benefits

As you can see from the previous section, mORMot provides a comprehensive set of features that can help you to manage your crosscutting concerns though a reusable set of components and core functionality.

Meet the mORMot

Of course, like many developers, you may suffer from the well-known NIH ("Not Invented Here") syndrome. On the other side, it is a commonly accepted fact that the use of standard and proven code libraries and components can save development time, minimize costs, reduce the use of precious test resources, and decrease the overall maintenance effort.

Benefits of mORMot are therefore:
- **KISS convention over configuration** design: you have all needed features at hand, but with only one way of doing it - less configuration and less confusion for the developer and its customers;
- **Pascal oriented**: implementation is not following existing Java or C# patterns (with generics (ab)use, variable syntaxes and black-box approach), but try to unleash the object pascal genius;
- **Integrated**: all crosscutting scenarios are coupled, so you benefit of consisting APIs and documentation, a lot of code-reuse, JSON/RESTful orientation from the ground up;
- **Tested**: most of the framework is test-driven, and all regression tests are provided, including system-wide integration tests;
- Do-not-reinvent-the-wheel, since we did it for you: it is now time to focus on your business;
- **Open Source**, documented and maintained: project is developed since years, with some active members - mORMot won't leave you soon!
1.4. Legacy code and existing projects

Even if mORMot will be more easily used in a project designed from scratch, it fits very well the purpose of evolving any existing Delphi project, or creating the server side part of an AJAX application. One benefit of such a framework is to facilitate the transition from a traditional Client-Server architecture to a N-Tier layered pattern.

Due to its modular design, you can integrate some framework bricks to your existing application:
- You may add logging to your code - see below (page 631), to track unresolved issues, and add customer-side performance profiling;
- Use low-level classes like record or dynamic array wrappers - see below (page 107), or our dynamic document storage via variant - see below (page 112), including JSON or binary persistence;
- You can use the direct DB layers, including the TQuery emulation class - see below (page 250) - to replace some BDE queries, or introduce nice unique features like direct database access or array binding for very fast data insertion - see below (page 239), or switch to a NoSQL database - see below (page 278);
- Reports could benefit of the mORMotReport.pas code-based system, which is very easy to use even on the server side (serving PDF files), when your business logic heavily relies on objects, not direct DB - see below (page 2519);
- HTTP requests may be made available using Client-Server services via methods - see below (page 373), e.g. for rendering HTML pages generated on the fly with Mustache templates - see below (page 505), pictures or PDF reports;
- You can little by little move your logic out of the client side code into some server services defined via interfaces, without the overhead of SOAP or WCF - see below (page 419); migration to SOA is the main benefit of mORMot for existing projects;
- Make your application ready to offer a RESTful interface, e.g. for consuming JSON content via AJAX or mobile clients - see below (page 481);
- New tables may be defined via the ORM/ODM features of mORMot, still hosted in your external SQL server - see below (page 239), as any previous data; in particular, mixed pure-ORM and regular-SQL requests may coexist; or mORMot’s data modeling may balance your storage among several servers (and technologies, like NoSQL);
- Sharing the same tables between legacy code SQL and mORMot ORM is possible, but to avoid consistency problems, you should better follow some rules detailed below (page 272);
- You may benefit from our very fast in-memory engine, a dedicated SQLite3-based consolidation database or even the caching features - see below (page 174), shared on the server side, when performance is needed - it may help integrating some CQRS pattern (Command Query Responsibility Segregation) into your application via a RESTful interface, and delegate some queries from your main database;
- If you are still using an old version of Delphi, and can’t easily move up due to some third party components or existing code base, mORMot will offer all the needed features to start ORM, N-Tier and SOA, starting with a Delphi 6 edition.

mORMot implements the needed techniques for introducing what Michael Feathers calls, in his book Working Effectively With Legacy Code, a seam. A seam is an area where you can start to cleave off some legacy code and begin to introduce changes. Even mocking abilities of mORMot - see below (page 406) - will help you in this delicate task - see http://www.infoq.com/articles/Utilizing-Logging..

Do not forget that Synopse, as a company, is able to offer dedicated audit and support for such a migration. The sooner, the better.
1.5. FAQ

Before you start going any further, we propose here below a simple FAQ containing the most frequent questions we received on our forums.

First of all, take a look at the keyword index available at the very beginning of this document. The underlined entries target the main article(s) about a given concept or technical term.

Feel free to give your feedback at https://synopse.info/forum, asking new questions or improving answers!

Your SAD doc is too long to read through in a short period.
Too much documentation can kill the documentation! But you do not need to read the whole document: most of it is a detailed description of every unit, object, or class. But the first part is worth reading, otherwise you are very likely to miss some main concepts or patterns. It just takes 15-30 minutes! Also read below (page 318) to find out in which direction you may need to go for writing your server code. Consider the slides available at https://drive.google.com/folderview?id=0B0r8u-FwvxWdeVJvZnBHseEpKye.

Where should I start?
Take a look at the Architecture principles below (page 81), then download and install the sources below (page 643), then compile and run the TestSQL3.dpr program. Check about ORM below (page 130), SOA below (page 419) and MVC below (page 519), then test the various samples (from the SQLite3\Samples folder), especially 01, 02, 04, 11, 12, 14, 17, 26, 28, 30 and the MainDemo.

So far, I can see your mORMot fits most of the requirement, but seems only for Database Client-Server apps.
First of all, the framework is a set of bricks, so you can use it e.g. to build interface based services, even with no database at all. We tried to make its main features modular and uncoupled.

I am not a great fan of ORM, sorry, I still like SQL and have some experience of that. Some times sophisticated SQL query is hard to change to ORM code.
ORM can make development much easier; but you can use e.g. interface-based services and "manual" SQL statements - in this case, you have at hand below (page 239) classes in mORMot, which allow very high performance and direct export to JSON.

I am tempted by using an ORM, but mORMot forces you to inherit from a root TSQLRecord type, whereas I'd like to use any kind of object.
We will discuss this in details below (page 192). Adding attributes to an existing class is tempting, but will pollute your code at the end, mixing persistence and business logic: see Persistence Ignorance and Aggregates below (page 599). The framework proposes a second level of Object mapping, allowing to persist any kind of PODO (Plain Old Delphi Object), by defining CQRS services - see below (page 610).

I would like to replace pieces of delphi-code by using mORMot and the DDD-concept in a huge system, but its legacy database doesn't have integer primary keys, and mORMot ORM expects a TID-like field.
By design, such legacy tables are not compatible with SQLite3 virtual tables, or our ORM - unless you add an ID integer additional primary key, which may not be the best idea. Some hints: write a persistence service as interface/class (as required by DDD - see below (page 599)); uncouple persistence and SOA services (i.e. the SOA TSQLRestServer is a TSQLRestServerFullMemory and not a DB/ORM TSQLRestServerDB); reuse your existing SQL statements, with SynDB as access layer if possible (you will have better performance, and direct JSON support); use the ORM for MicroService
local persistence (with SQLite3), and/or for new tables in your legacy DB (or another storage, e.g. MongoDB).

**Why are you not using the latest features of the compiler, like generics or class attributes?**

Our framework does not rely on generics, but on the power of the object pascal type system: specifying a class or interface type as parameter is safe and efficient - and generics tends to blow the executable size, lower down performance (the current RTL is not very optimized, and sometimes bugged), and hide implementation details. Some methods are available for newer version of the compiler, introducing access via generics; but it was not mandatory to depend on them. We also identified, as several Java or C# gurus, that class attributes may sound like a good idea, but tend to pollute the code, and introduce unexpected coupling. Last but not least, those features are incompatible with older version of Delphi we would like to support, and may reduce compatibility with FPC.

I also notice in your SAD doc, data types are different from Delphi. You have RawUTF8, etc, which make me puzzled, what are they?

You can for sure use standard Delphi string types, but some more optimized types were defined: since the whole framework is UTF-8 based, we defined a dedicated type, which works with all versions of Delphi, before and after Delphi 2009. By the way, just search for RawUTF8 in the keyword index of this document, or see below (page 105).

**During my tests, my client receives non standard JSON with unquoted fields.**

Internally, the framework uses JSON in MongoDB extended syntax, i.e. fields are not quoted - this gives better performance and reduces memory and bandwidth with a mORMot client. To receive "field":value instead of field:value, just add a proper User-Agent HTTP header to the client request (as any browser does), and the server will emit standard JSON.

**When I work with floating points and JSON, sometimes numerical values with more than 4 decimals are converted into JSON strings.**

By default, double values are disabled in the JSON serialization, to avoid any hidden precision lost during conversion: see below (page 119) how to enable it.

I got an access violation with SynDB ISQLDBRows.

You need to explicitly release the ISQLDBRows instance, by setting it to nil, before freeing the owner's connection - see below (page 248).

**Deadlock occurs with interface callbacks.**

When working with asynchronous notifications over WebSockets, you need to ensure you won't fire directly a callback from a main method execution - see below (page 453) for several solutions.

**All the objects seem non-VCL components, meaning need code each property and remember them all well.**

This is indeed... a feature. The framework is not RAD, but fully object-oriented. Thanks to the Delphi IDE, you can access all properties description via auto-completion and/or code navigation. We tried to make the documentation exhaustive and accurate. Then you can still use RAD for UI design, but let business be abstracted in pure code. See e.g. the mORMotVCL.pas unit which can publish any ORM result as TDataSource for your UI.

I know you have joined the DataSnap performance discussion and your performance won good reputation there. If I want to use your framework to replace my old project of DataSnap, how easy will it be?

If you used DataSnap to build method-based services, translation into mORMot will be just a matter of code refactoring. And you will benefit of new features like Interface-based services - see below (page 453) for several solutions.
which is much more advanced than the method-based pattern, and will avoid generating the client class via a wizard, and offers additional features - see below (page 459) or below (page 461).

What is the SMS? Do you know any advantage compared to JQuery or AngularJS?

*Smart Mobile Studio* is an IDE and some source runtime able to develop and compile an Object-Pascal project into a *HTML 5 / CSS 3 / JavaScript embedded* application, i.e. able to work stand alone with no remote server. When used with *mORMot* on the server side, you can use the very same object pascal language on both server and client sides, with strong typing and true OOP design. Then you feature secure authentication and JSON communication, with connected or off-line mode. Your *SmartPascal* client code can be generated by your *mORMot* server, as stated below (page 486). We currently focus on TMS Web Core integration, which seems a newer - and more supported - alternative.

I am trying to search a substitute solution to WebSnap. Do you have any sample or doc to describe how to build a robust web Server?

You can indeed easily create a modern MVC / MVVM scaling Web Application. Your *mORMot* server can easily publish its ORM / SOA business logic as *Model*, use *Mustache* logic-less templates rendering - see below (page 505) - for *Views*, and defining the *ViewModel / Controller* as regular Delphi methods. See below (page 519) for more details, and discovering a sample "blog" application.

Have you considered using a popular source coding host like Github or BitBucket?

We love to host our own source code repository, and find fossil a perfect match for our needs, with a friendly approach. But we created a parallel repository on *GitHub*, so that you may be able to monitor or fork our projects - see [https://github.com/synopse/mORMot](https://github.com/synopse/mORMot). Note that you can get a daily snapshot of our official source code repository directly from [https://synopse.info/files/mORMotNightlyBuild.zip](https://synopse.info/files/mORMotNightlyBuild.zip).

Why is this framework named *mORMot*?

- Because its initial identifier was "*Synopse SQLite3 database framework*", which may induce a *SQLite3*-only library, whereas the framework is now able to connect to any database engine;
- Because we like mountains, and those large ground rodents;
- Because marmots do hibernate, just like our precious objects;
- Because marmots are highly social and use loud whistles to communicate with one another, just like our applications are designed not to be isolated;
- Because even if they eat greens, they use to fight at Spring for their realm;
- Because it may be an acronym for "Manage Object Relational Mapping Over Territory", or whatever you may think of...
2. Architecture principles

This framework tries to implement some "best-practice" patterns, among them:

- Model-View Controller - see below (page 86);
- Multi-tier architecture - see below (page 88);
- Test-Driven Design - see below (page 626);
- Stateless CRUD/REST - see below (page 311);
- Object-Relational Mapping - see below (page 92);
- Object-Document Mapping - see below (page 96);
- Service-Oriented Architecture - see below (page 90).

All those points render possible any project implementation, up to complex Domain-Driven design - see below (page 99).
2.1. General design

A general design of the mORMot architecture is shown in the following diagram:

General mORMot architecture - Client Server implementation

In addition, you may use the following transversal features:

General mORMot architecture - Cross-Cutting features

Don’t be afraid. Such a drawing may sound huge and confusing, especially when you have a RAD background, and did not work much with modern design patterns.
Following pages will detail and explain how the framework implements this architecture, and sample code is available to help you discovering the amazing mORMot realm.

In the previous diagram, you can already identify some key concepts of mORMot:
- Cross-Platform, multi clients, and multi devices;
- Can integrate to an existing code base or architecture;
- Client-Server RESTful design;
- Layered (multi-tier) implementation;
- Process can be defined via a set of Services (SOA);
- Business rules and data model are shared by Clients and Server;
- Data is mapped by objects (ORM/ODM);
- Databases can be an embedded SQLite3, one or several standard RDBMS (with auto-generated SQL), a MongoDB NoSQL engine, fast in-memory objects lists, or another mORMot server;
- Security (authentication and authorization) is integrated to all layers;
- User interface and reporting classes are available;
- You can write a MVC/MVVM AJAX or Web Application from your ORM/SOA methods;
- Based on simple and proven patterns (REST, JSON, MVC, SOLID);
- A consistent testing and debugging API is integrated;
- Optimized for both scaling and stability.
2.2. Architecture Design Process

First point is to state that you can't talk about architecture in isolation. Architecture is always driven by the actual needs of the application, not by whatever the architect read about last night and is dying to see how it works in the real world. There is no such "one architecture fits all" nor "one framework fits all" solution. Architecture is just a thinking of how you are building your own software.

In fact, software architecture is not about theory and diagrams, nor just about best practice, but about a way of implementing a working solution for your customers.

![Architecture Iterative Process (SCRUM)](image)

This diagram presents how Architecture is part of a typical SCRUM iterative agile process. Even if some people of your company may be in charge of global software architecture, or even if your project managements follows a classic V-cycle and does not follow the agile manifesto, architecture should never be seen as a set of rules, to be applied by every and each developers. Architecture is part of the coding, but not all the coding.

Here are some ways of achieving weak design:
- Let each developer decides, from his/her own knowledge (and mood?), how to implement the use cases, with no review, implementation documentation, nor peer collaboration;
- Let each team decides, from its own knowledge (and untold internal leadership?), how to implement the use cases, with no system-wide collaboration;
- Let architecture be decided at so high level that it won't affect the actual coding style of the developers (just don't be caught);
- Let architecture be so much detailed that each code line has to follow a typical implementation
pattern, therefore producing over engineered code;
- Let architecture map the existing, with some middle-term objectives at best;
- Let technology, frameworks or just-blogged ideas be used with no discrimination (do not trust the sirens of dev marketing).

Therefore, some advices:
- Collaboration is a need - no one is alone, no team is better, no manager is always right;
- Sharing is a need - between individuals, as teams, with managers;
- Stay customer and content focused;
- Long term is prepared by today's implementation;
- Be lazy, i.e. try to make tomorrow's work easier for you and your team-workers;
- They did not know it was impossible, so they did it.

Purpose of frameworks like mORMot is to provide your teams with working and integrated set of classes, so that you can focus on your product, enjoying the collaboration with other Open Source users, in order to use evolving and pertinent software architecture.
2.3. Model-View-Controller

The Model-View-Controller (MVC) is a software architecture, currently considered an architectural pattern used in software engineering. The pattern isolates "domain logic" (the application logic for the user) from the user interface (input and presentation), permitting independent development, testing and maintenance of each (separation of concerns).

The Model manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller). In Event-Driven systems, the model notifies observers (usually views) when the information changes so that they can react - but since our ORM is stateless, it does not need to handle those events - see below (page 314).

The View renders the model into a form suitable for interaction, typically a user interface element. Multiple views can exist for a single model for different purposes. A viewport typically has a one to one correspondence with a display surface and knows how to render to it.

The Controller receives user input and initiates a response by making calls on model objects. A controller accepts input from the user and instructs the model and viewport to perform actions based on that input.

In the framework, the model is not necessarily merely a database; the model in MVC is both the data and the business/domain logic needed to manipulate the data in the application. In our ORM, a model is implemented via a TSQLModel class, which centralizes all TSQLRecord inherited classes used by an application, both database-related and business-logic related.

The views can be implemented using:
- For Desktop clients, a full set of User-Interface units of the framework, which is mostly auto-generated from code - they will consume the model as reference for rendering the data;
- For Web clients, an integrated high-speed Mustache rendering engine - see below (page 505) - is able to render HTML pages with logic-less templates, and controller methods written in Delphi - see below (page 519);
- For AJAX clients, the server side is easy to be reached from RESTful JSON services.

The controller is mainly already implemented in our framework, within the RESTful commands, and will interact with both the associated view (e.g. for refreshing the User Interface) and model (for data handling). Some custom actions, related to the business logic, can be implemented via some custom TSQLRecord classes or via custom RESTful Services - see below (page 372).
2.4. Multi-tier architecture

In software engineering, multi-tier architecture (often referred to as n-tier architecture) is a client–server architecture in which the presentation, the application processing, and the data management are logically separate processes. For example, an application that uses middleware to service data requests between a user and a database employs multi-tier architecture. The most widespread use of multi-tier architecture is the three-tier architecture.

In practice, a typical VCL/FMX RAD application written in Delphi has a two-tier architecture:

![Two-Tier Architecture - Logical View](image)

**Two-Tier Architecture - Logical View**

In this approach, the *Application Tier* mixes the UI and the logic in forms and modules.

Both ORM and SOA aspects of our RESTful framework make it easy to develop using a more versatile three-tier architecture.

![Multi-Tier Architecture - Logical View](image)

**Multi-Tier Architecture - Logical View**

The *Synops e mORMot Framework* follows this development pattern:

- **Data Tier** is either SQLite3 and/or an internal very fast in-memory database; most SQL queries are created on the fly, and database table layout are defined from *Delphi* classes; you can also use any external database, currently SQLite3, Oracle, Jet/MSAccess, MS SQL, Firebird, DB2, PostgreSQL, MySQL, Informix and NexusDB SQL dialects are handled, and even NoSQL engines like *MongoDB* can be directly used - see below (page 239);

- **Logic Tier** is performed by pure ORM aspect and SOA implementation: you write *Delphi* classes which are mapped by the *Data Tier* into the database, and you can write your business logic as Services called as *Delphi* interface, up to a Domain-Driven design - see below (page 99) - if your project reaches some level of complexity;

- **Presentation Tier** is either a *Delphi* Client, or an AJAX application, because the framework can communicate using RESTful JSON over HTTP/1.1 (the *Delphi* Client User Interface is generated from Code, by using RTTI and structures, not as a RAD - and the Ajax applications need to be written by using your own tools and JavaScript framework, there is no "official" Ajax framework included yet).

In fact, *mORMot* can scales up to a Domain-Driven Design four-tier architecture - see below (page 99) - as such:
- **Presentation Tier** which can be e.g. a Delphi or AJAX client;
- **Application Tier** which serves JSON content according to the client application;
- **Business Logic Tier** which centralizes all the Domain processing, shared among all applications;
- **Persistence/Data Tier** which can be either in-process (like SQLite3 or in-memory) or external (e.g. Oracle, MS SQL, DB2, PostgreSQL, MySQL, Informix...).

---

**Domain Driven Design n-Tier Architecture - Logical View**

Note that you have to make a difference between physical and logical n-tier architecture. Most of the time, n-Tier is intended to be a physical (hardware) view, for instance a separation between the database server and the application server, placing the database on a separate machine to facilitate ease of maintenance. In mORMot, and more generally in SOA - see below (page 90), we deal with logical layout, with separation of layers through interfaces - see below (page 385) - and the underlying hardware implementation will usually not match the logical layout.

---

**Domain Driven Design n-Tier Architecture - Physical View**

In this document, we will focus on the logical way of thinking / coding, letting the physical deployment be made according to end-user expectations.
2.5. Service-Oriented Architecture (SOA)

Service-Oriented Architecture (SOA) is a flexible set of design principles used during the phases of systems development and integration in computing. A system based on a SOA will package functionality as a suite of inter-operable services that can be used within multiple, separate systems from several business domains.

A software service is a logical representation of a repeatable activity that produce a precise result. In short, a consumer ask to a producer to act in order to produce a result. Most of the time, this invocation is free from any previous invocation (it is therefore called stateless).

The SOA implementations rely on a mesh of software services. Services comprise unassociated, loosely coupled units of functionality that have no calls to each other embedded in them. Each service implements one action, such as filling out an online application for an account, or viewing an online bank statement, or placing an online booking or airline ticket order. Rather than services embedding calls to each other in their source code, they use defined protocols that describe how services pass and parse messages using description meta-data.

Since most of those services are by definition stateless, some kind of service composition is commonly defined to provide some kind of logical multi-tier orchestration of services. A higher level service invokes several services to work as a self-contained, stateless service; as a result, lower-level services can still be stateless, but the consumer of the higher level service is able to safely process some kind of transactional process.


SOA is mainly about decoupling.
That is, it enables implementation independence in a variety of ways, for instance:
### Dependency | Desired decoupling | Decoupling technique
--- | --- | ---
**Platform** | Hardware, Framework or Operating System should not constrain choices of the Services consumers | Standard protocols, mainly Web services (e.g. SOAP or RESTful/JSON)
**Location** | Consumers may not be impacted by service hosting changes | Routing and proxies will maintain Services access
**Availability** | Maintenance tasks shall be transparent | Remote access allows centralized support on Server side
**Versions** | New services shall be introduced without requiring upgrades of clients | Contract marshalling can be implemented on the Server side

SOA and ORM - see below (page 92) - do not exclude themselves. In fact, even if some software architects tend to use only one of the two features, both can coexist and furthermore complete each other, in any Client-Server application:

- ORM access could be used to access to the data with objects, that is with the native presentation of the Server or Client side (Delphi, JavaScript...) - so ORM can be used to provide efficient access to the data or the business logic - this is the idea of CQRS pattern;
- SOA will provide a more advanced way of handling the business logic: with custom parameters and data types, it is possible to provide some high-level Services to the clients, hiding most of the business logic, and reducing the needed bandwidth.

In particular, SOA will help leaving the business logic on the Server side, therefore will help increasing the Multi-tier architecture (page 88). By reducing the back-and-forth between the Client and the Server, it will also reduce the network bandwidth, and the Server resources (it will always cost less to run the service on the Server than run the service on the Client, adding all remote connection and serialization to the needed database access). Our interface-based SOA model allows the same code to run on both the client and the server side, with a much better performance on the server side, but a full interoperability of both sides.
2.6. Object-Relational Mapping (ORM)

In practice, ORM gives a set of methods to ease high-level objects persistence into a RDBMS.

Our Delphi class instances are not directly usable with a relational database, which is since decades the most convenient way of persisting data. So some kind of "glue" is needed to let class properties be saved into one or several tables. You can interact with the database using its native language, aka SQL. But SQL by itself is a full programming language, with diverse flavors depending on the exact backend engine (just think about how you define a column type able to store text). So writing and maintaining your SQL statements may become a time-consuming, difficult and error-prone task.

Sometimes, there will be nothing better than a tuned SQL statement, able to aggregate and join information from several tables. But most of the time, you will need just to perform some basic operations, known as CRUD (for Create Retrieve Update Delete actions) on well identified objects: this is where ORM may give you a huge hint, since it is able to generate the SQL statements for you.

The ORM works in fact as such:

The ORM core retrieve information to perform the mapping:
- Object definition via its class type (via RTTI);
  - Database model as retrieved for each database engine.
Since several implementation schemes are possible, we will first discuss the pros and the cons of each one.

First, here is a diagram presenting some common implementation schemes of database access with Delphi (which maps most other languages or frameworks, including C# or Java).

Why a Client-Server ORM
The table below is a very suggestive (but it doesn’t mean wrong) Resumé of some common schemes, in the Delphi world. ORM is just one nice possibility among others.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use DB views and tables,</td>
<td>- SQL is a powerful language</td>
<td>- Business logic can’t be elaborated without stored procedures</td>
</tr>
<tr>
<td>with GUI components</td>
<td>- Can use high-level DB tools (UML) and RAD approach</td>
<td>- SQL code and stored procedures will bind you to a DB engine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Poor Client interaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Reporting must call the DB directly</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- No Multi-tier architecture</td>
</tr>
<tr>
<td>Map DB tables or views with Delphi</td>
<td>- Can use elaborated business logic, in Delphi</td>
<td>- SQL code must be coded by hand and synchronized with the classes</td>
</tr>
<tr>
<td>classes</td>
<td>- Separation from UI and data</td>
<td>- Code tends to be duplicated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- SQL code could bind you to a DB engine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Reports can be made from code or via DB related tools</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Difficult to implement true Multi-tier architecture</td>
</tr>
<tr>
<td>Use a Database ORM</td>
<td>- Can use very elaborated business logic, in Delphi</td>
<td>- More abstraction needed at design time (no RAD approach)</td>
</tr>
<tr>
<td></td>
<td>- SQL code is generated (in most cases) by the ORM</td>
<td>- In some cases, could lead to retrieve more data from DB than needed</td>
</tr>
<tr>
<td></td>
<td>- ORM will adapt the generated SQL to the DB engine</td>
<td>- Not yet a true Multi-tier architecture, because ORM is for DB access only and business logic will need to create separated classes</td>
</tr>
<tr>
<td>Use a Client-Server ORM</td>
<td>- Can use very elaborated business logic, in Delphi</td>
<td>- More abstraction needed at design time (no RAD approach)</td>
</tr>
<tr>
<td></td>
<td>- SQL code is generated (in most cases) by the ORM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- ORM will adapt the generated SQL to the DB engine</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Services will allow to retrieve or process only needed data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Server can create objects viewed by the Client as if they were DB objects, even if they are only available in memory or the result of some business logic defined in Delphi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Complete Multi-tier architecture</td>
<td></td>
</tr>
</tbody>
</table>

Of course, you’ll find out that our framework implements a Client-Server ORM, which can be down-sized to stand-alone mode if needed, but which is, thanks to its unique implementation, scalable to any complex Domain-Driven Design.

As far as we found out, looking at every language and technology around, almost no other ORM supports such a native Client-Server orientation. Usual practice is to use a Service-Oriented
Architecture (SOA) (page 90) for remote access to the ORM. Some projects allow remote access to an existing ORM, but they are separated projects. Our mORMot is pretty unique, in respect to its RESTful Client-Server orientation, from the ground up.

If you entered the Delphi world years ago, you may be pretty fluent with the RAD approach. But you probably also discovered how difficult it is to maintain an application which mixes UI components, business logic and database queries. Today's software users have some huge ergonomic expectations about software usability: some screens with grids and buttons, mapping the database, won't definitively be appealing. Using mORMot's ORM /SOA approach will help you focus on your business and your clients expectations, letting the framework perform most of the plumbing for you.
2.7. NoSQL and Object-Document Mapping (ODM)

SQL is the De-Facto standard for data manipulation
- Schema-based;
- Relational-based;
- ACID by transactions;
- Time proven and efficient;
- "Almost" standard (each DB has its own column typing system).

NoSQL is a new paradigm, named as such in early 2009 (even if some database engines, like Lotus Domino, may fit the definition since decades):
- NoSQL stands for "Not Only SQL" - which is more positive than "no SQL";
- Designed to scale for the web and BigData (e.g. Amazon, Google, Facebook), e.g. via easy replication and simple API;
- Relying on no standard (for both data modeling and querying);
- A lot of diverse implementations, covering any data use - http://nosql-database.org lists more than 150 engines.

We can identify two main families of NoSQL databases:
- Graph-oriented databases;
- Aggregate-oriented databases.

Graph-oriented databases store data by their relations / associations:

NoSQL Graph Database

Such kind of databases are very useful e.g. for developing any "social" software, which will value its data by the relations between every node. Such data model does not fit well with the relational model, whereas a NoSQL engine like Neo4j handles such kind of data natively. Note that by design, Graph-oriented databases are ACID.

But the main NoSQL database family is populated by the Aggregate-oriented databases. By Aggregate,
we mean the same definition as will be used below (page 99) for *Domain Driven Design*. It is a collection of data that we interact with as a unit, which forms the boundaries for ACID operations in a given model.

In fact, *Aggregate*-oriented databases can be specified as three main implementation/query patterns:
- Document-based (e.g. *MongoDB*, *CouchDB*, *RavenDB*);
- Key/Value (e.g. *Redis*, *Riak*, *Voldemort*);
- Column family (e.g. *Cassandra*, *HiBase*).

Some of them can be *schema-less* (meaning that the data layout is not fixed, and can evolve on the fly without re-indexing the whole database) - but column-driven bases do have a schema, or even storing plain BLOB of data (this is the purpose of Key/Value engines, which focus on storage speed and rely on the client side to process the data).

In short, RDBMS stores data per table, and need to JOIN the references to get the aggregated information:

![SQL Aggregate via JOINed tables](image)

Whereas *NoSQL* stores its aggregates as documents: the whole data is embedded in one.

![NoSQL Aggregate as one document](image)

Which may be represented as the following JSON - see below (page 295) - data:

```json
{
  "ID": 1234,
  "UserName": "John Smith",
  "Contact": {
    "Phone": "123-456-789",
    "Email": "xyz@abc.com"
  },
  "Access": {
    "Level": 5,
    "Group": "dev"
  }
}
```

Such a document will fit directly with the object programming model, without the need of thinking about JOINed queries and database plumbing.

As a result, we can discuss the two data models:
- **Relational data Model** with highly-structured table organization, and rigidly-defined data formats and record structure;
- **Document data Model** as a collection of complex documents with arbitrary, nested data formats and varying "record" format.

The *Relational* model features *normalization* of data, i.e. organize the fields and tables of a relational database to minimize redundancy.

On the other hand, the *Document* model features *denormalization* of data, to optimize the read performance of a database by adding redundant data or by grouping data. It also features horizontal scaling of the servers, since data can easily be balanced among several servers, without the speed penalty of performing a remote JOIN.

One of the main difficulties, when working with *NoSQL*, is to define how to *denormalize* the data, and when to store the data in *normalized* format.

One good habit is to model your data depending on the most current queries you will have to perform. For instance, you may embed sub-documents which will be very likely to be requested by your application most of the time. Note that most *NoSQL* engines feature a *projection* mechanism, which allows you to return only the needed fields for a query, leaving the sub-documents on the server if you do not need them at this time. The less frequent queries may be executed over separated collections, populated e.g. with consolidated information.

Since *NoSQL* databases have fewer hard-and-fast rules than their relational databases ancestors, you are more likely to tune your model, depending on your expectations. In practice, you may spend less time thinking about "how" to store the data than with a RDBMS, and are still able to *normalize* information later, if needed. *NoSQL* engines do not fear redundant information, as soon as you follow the rules of letting the client application take care of the whole data consistency (e.g. via one ORM).

As you may have stated, this *Document data Model* is much closer to the OOP paradigm than the classic relational scheme. Even a new family of frameworks did appear together with *NoSQL* adoption, named *Object Document Mapping* (ODM), which is what *Object-Relational Mapping* (ORM) (page 92) was for RDBMS.

In short, both approaches have benefits, which are to be weighted.

<table>
<thead>
<tr>
<th>SQL</th>
<th>NoSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubiquitous SQL</td>
<td>Map OOP and complex types (e.g. arrays or nested documents)</td>
</tr>
<tr>
<td>Easy vertical scaling</td>
<td>Uncoupled data: horizontal scaling</td>
</tr>
<tr>
<td>Data size (avoid duplicates and with no schema)</td>
<td>Schema-less: cleaner evolution</td>
</tr>
<tr>
<td>Data is stored once, therefore consistent</td>
<td>Version management (e.g. CouchDB)</td>
</tr>
<tr>
<td>Complex ACID statements</td>
<td>Graph storage (e.g. Redis)</td>
</tr>
<tr>
<td>Aggregation functions (depends)</td>
<td>Map/Reduce or Aggregation functions (e.g. since MongoDB 2.2)</td>
</tr>
</tbody>
</table>

With mORMot, you can switch from a classic SQL engine into a trendy *MongoDB* server, just in one line of code, when initializing the data on the server side. You can switch from ORM to ODM at any time, even at runtime, e.g. for a demanding customer.
2.8. Domain-Driven Design

2.8.1. Definition

http://domaindrivendesign.org.. gives the somewhat "official" definition of Domain-Driven design (DDD):

Over the last decade or two, a philosophy has developed as an undercurrent in the object community. The premise of domain-driven design is two-fold:

- For most software projects, the primary focus should be on the domain and domain logic;
- Complex domain designs should be based on a model.

Domain-driven design is not a technology or a methodology. It is a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains.

Of course, this particular architecture is customizable according to the needs of each project. We simply propose following an architecture that serves as a baseline to be modified or adapted by architects according to their needs and requirements.

2.8.2. Patterns

In respect to other kinds of Multi-tier architecture (page 88), DDD introduces some restrictive patterns, for a cleaner design:

- Focus on the Domain - i.e. a particular kind of knowledge;
- Define Bounded contexts within this domain;
- Create an evolving Model of the domain, ready-to-be consumed by applications;
- Identify some kind of objects - called Value objects or Entity Objects / Aggregates;
- Use an Ubiquitous Language in resulting model and code;
- Isolate the domain from other kind of concern (e.g. persistence should not be called from the domain layer - i.e. the domain should not be polluted by technical considerations, but rely on the Factory and Repository patterns);
- Publish the domain as well-defined uncoupled Services;
- Integrate the domain services with existing applications or legacy code.

The following diagram is a map of the patterns presented and the relationships between them. It is inspired from the one included in the Eric Evans's reference book, "Domain-Driven Design", Addison-Wesley, 2004 (and updated to take in account some points appeared since).
Domain-Driven Design - Building Blocks

You may recognize a lot of existing patterns you already met or implemented. What makes DDD unique is that those patterns have been organized around some clear concepts, thanks to decades of business software experiment.

2.8.3. Is DDD good for you?

Domain-Driven design is not to be used everywhere, and in every situation.

First of all, the following are prerequisite of using DDD:

- Identified and well-bounded domain (e.g. your business target should be clearly identified);
- You must have access to domain experts to establish a creative collaboration, in an iterative (may be agile) way;
- Skilled team, able to write clean code - note also that since DDD is more about code expressiveness than technology, it may not appear so "trendy" to youngest developers;
- You want your internal team to accumulate knowledge of the domain - therefore, outsourcing may be constrained to applications, not the core domain.

Then check that DDD is worth it, i.e. if:

- It helps you solving the problem area you are trying to address;
- It meets your strategic goals: DDD is to be used where you will get your business money, and make you distinctive from your competitors;
- You need to bring clarity, and need to solve inner complexity, e.g. modeling a lot of rules (you won’t use DDD to build simple applications - in this case, RAD may be enough);
- Your business is exploring: your goal is identified, but you do not know how to accomplish it;
- Don’t have all of these concerns, but at least one or two.

2.8.4. Introducing DDD

Perhaps DDD sounds more appealing to you now. In this case, our mORMot framework will provide all the bricks you need to implement it, focusing on your domain and letting the libraries do all the needed plumbing.

If you identified that DDD is not to be used now, you will always find with mORMot the tools you need, ready to switch to DDD when it will be necessary.

Legacy code and existing projects (page 77) will benefit from DDD patterns. Finding so-called seams, along with isolating your core domain, can be extremely valuable when using DDD techniques to refactor and tighten the highest value parts of your code. It is not mandatory to re-write your whole existing software with DDD patterns everywhere: once you have identified where your business strategy’s core is, you can introduce DDD progressively in this area. Then, following continuous feedback, you will refine your code, adding regression tests, and isolating your domain code from end-user code.

For a technical introduction about DDD and how mORMot can help you implement this design, see below (page 595).

With mORMot, your software solution will never be stuck in a dead-end. You’ll be able to always adapt to your customers need, and maximize your ROI.
3. Enter new territory

3.1. Meet the mORMot

The Synopse mORMot framework consists in a huge number of units, so we will start by introducing them.
### 3.2. Main units

The main units you have to be familiar with are the following:

<table>
<thead>
<tr>
<th>Unit name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons.pas</td>
<td>Common types, classes and functions</td>
</tr>
<tr>
<td>SynLog.pas</td>
<td></td>
</tr>
<tr>
<td>SynTests.pas</td>
<td></td>
</tr>
<tr>
<td>mORMot.pas</td>
<td>Main unit of the ORM / SOA framework</td>
</tr>
<tr>
<td>SynSQLite3.pas</td>
<td><strong>SQLite3</strong> database engine</td>
</tr>
<tr>
<td>SynSQLite3Static.pas</td>
<td></td>
</tr>
<tr>
<td>mORMotSQLite3.pas</td>
<td>Bridge between mORMot.pas and SynSQLite3.pas</td>
</tr>
<tr>
<td>SynDB.pas</td>
<td>Direct RDBMS access classes</td>
</tr>
<tr>
<td>SynDB*.pas</td>
<td></td>
</tr>
<tr>
<td>mORMotDB.pas</td>
<td>ORM external SynDB.pas access, via <strong>SQLite3</strong> virtual tables</td>
</tr>
<tr>
<td>SynMongoDB.pas</td>
<td>Direct access to a <strong>MongoDB</strong> server</td>
</tr>
<tr>
<td>mORMotMongoDB.pas</td>
<td></td>
</tr>
<tr>
<td>SynSM.pas</td>
<td><strong>SpiderMonkey</strong> JavaScript engine</td>
</tr>
<tr>
<td>SynSMAPI.pas</td>
<td></td>
</tr>
<tr>
<td>mORMotHttpClient.pas</td>
<td>RESTful HTTP/1.1 Client and Server</td>
</tr>
<tr>
<td>mORMotHttpServer.pas</td>
<td></td>
</tr>
<tr>
<td>SynCrtSock.pas</td>
<td></td>
</tr>
<tr>
<td>mORMotMVC.pas</td>
<td>MVC classes for writing Web Applications</td>
</tr>
<tr>
<td>SynMustache.pas</td>
<td></td>
</tr>
<tr>
<td>mORMotUI*.pas</td>
<td>Grid and Forms User Interface generation</td>
</tr>
<tr>
<td>mORMotToolBar.pas</td>
<td>ORM ToolBar User Interface generation</td>
</tr>
<tr>
<td>mORMotReport.pas</td>
<td>Integrated Reporting engine</td>
</tr>
</tbody>
</table>

Other units are available in the framework source code repository, but are either expected by those files above (e.g. like SynDB*.pas database providers), or used only optionally in end-user cross-platform client applications (e.g. the CrossPlatform folder).

In the following pages, the features offered by those units will be presented.

Do not forget to take a look at all sample projects available in the **SQLite3\Samples** sub-folders - nothing is better than some simple code to look at.

Then detailed information will be available in the second part of this document - see below (page 643).
4. SynCommons unit

First of all, let us introduce some cross-cutting features, used everywhere in the Synopse source code. Even if you do not need to go deeply into the implementation details, it will help you not be disturbed with some classes and types you may encounter in the framework source, and its documentation.

It was a design choice to use some custom low-level types, classes and functions instead of calling the official Delphi RTL. Benefits could be:
- Cross-platform and cross-compiler support (e.g. leverage specificities, about memory model or RTTI);
- Unicode support for all versions of Delphi, even before Delphi 2009, or with FPC;
- Optimized for process speed, multi-thread friendliness and re-usability;
- Sharing of most common features (e.g. for text/data processing);
- KISS and consistent design.

In order to use Synopse mORMot framework, you should better be familiar with some of those definitions.

First of all, a Synopse.inc include file is provided, and appears in most of the framework units:

```
{$I Synopse.inc} // define HASINLINE USETYPEINFO CPU32 CPU64
```

It will define some conditionals, helping write portable and efficient code.

In the following next paragraphs, we'll comment some main features of the lowest-level part of the framework, mainly located in SynCommons.pas:
- Unicode and UTF-8;
- Currency type;
- Dynamic array wrappers (TDynArray and TDynArrayHashed);
- TDocVariant custom variant type for dynamic schema-less object or array storage.

Other shared features available in SynTests.pas and SynLog.pas will be detailed later, i.e. Testing and Logging - see below (page 626).
4.1. Unicode and UTF-8

Our mORMot Framework has 100% UNICODE compatibility, that is compilation under Delphi 2009 and up (including latest Delphi 10.3 Rio revision). The code has been deeply rewritten and tested, in order to provide compatibility with the String=UnicodeString paradigm of these compilers. But the code will also handle safely Unicode for older versions, i.e. from Delphi 6 up to Delphi 2007.

From its core to its uppermost features, our framework is natively UTF-8, which is the de-facto character encoding for JSON, SQLite3, and most supported database engines. This allows our code to offer fast streaming/parsing in a SAX-like mode, avoiding any conversion between encodings from the storage layer to your business logic. We also needed to establish a secure way to use strings, in order to handle all versions of Delphi (even pre-Unicode versions, especially the Delphi 7 version we like so much), and provide compatibility with the FreePascal Compiler. This consistency allows to circumvent any RTL bug or limitation, and ease long-term support of your project.

Some string types have been defined, and used in the code for best cross-compiler efficiency:
- RawUTF8 is used for every internal data usage, since both SQLite3 and JSON do expect UTF-8 encoding;
- WinAnsiString where WinAnsi-encoded AnsiString (code page 1252) are needed;
- Generic string for i18n (e.g. in unit mORMoti18n), i.e. text ready to be used within the VCL, as either AnsiString (for Delphi 2 to 2007) or UnicodeString (for Delphi 2009 and later);
- RawUnicode in some technical places (e.g. direct Win32 *W() API call in Delphi 7) - note: this type is NOT compatible with Delphi 2009 and later UnicodeString;
- RawByteString for byte storage (e.g. for FileFromString() function);
- SynUnicode is the fastest available Unicode native string type, depending on the compiler used (i.e. WideString before Delphi 2009, and UnicodeString since);
- Some special conversion functions to be used for Delphi 2009+ UnicodeString (defined inside {$ifdef UNICODE}...{$endif} blocks);
- Never use AnsiString directly, but one of the types above.

Note that RawUTF8 is the preferred string type to be used in our framework when defining textual properties in a TSQLRecord and for all internal data processing. It is only when you're reaching the User Interface layer that you may convert explicitly the RawUTF8 content into the generic VCL string type, using either the Language. UTF8ToString method (from mORMoti18n.pas unit) or the following function from SynCommons.pas:

```pascal
function UTF8ToString(const Text: RawUTF8): string;
```

Of course, the StringToUTF8 method or function are available to send back some text to the ORM layer.

A lot of dedicated conversion functions (including to/from numerical values) are included in SynCommons.pas. Those were optimized for speed and multi-thread capabilities, and to avoid implicit conversions involving a temporary string variable.

Warning during the compilation process are not allowed, especially under Unicode version of Delphi (e.g. Delphi 2010): all string conversion from the types above are made explicitly in the framework's code, to avoid any unattended data loss.
If you are using older version of Delphi, and have an existing code base involving a lot of WideString variables, you may take a look at the SynFastWideString.pas unit. Adding this unit in the top of your .dpr uses clauses will let all WideString process use the Delphi heap and its very efficient FastMM4 memory manager, instead of the much slower BSTR Windows API. Performance gain can be more than 50 times, if your existing code uses a lot of WideString variables. Note that using this unit will break the compatibility with BSTR/COM/OLE kind of string, so is not to be used with COM objects. In all cases, if you need Unicode support with older versions of Delphi, consider using our RawUTF8 type instead, which is much better integrated with our framework, and has less overhead.

### 4.2. Currency handling

Faster and safer way of comparing two currency values is certainly to map the variables to their internal Int64 binary representation, as such:

```pascal
function CompCurrency(A,B: currency): Int64;
var
  A64: Int64 absolute A;
  B64: Int64 absolute B;
begin
  result := A64-B64;
end;
```

This will avoid any rounding error during comparison (working with *10000 integer values), and is likely to be faster than the default implementation, which uses the FPU (or SSE2 under x64 architecture) instructions.

Some direct currency processing is available in the SynCommons.pas unit. It will by-pass the FPU use, and is therefore very fast.

There are some functions using the Int64 binary representation (accessible either as PInt64(@aCurrencyVar)^ or the absolute syntax):

- function Curr64ToString(Value: Int64): string;
- function StrToCurr64(P: PUTF8Char): Int64;
- function Curr64ToStr(Value: Int64): RawUTF8;
- function Curr64ToPChar(Value: Int64; Dest: PUTF8Char): PInt;
- function StrCurr64(P: PAnsiChar; const Value: Int64): PAnsiChar;

Using those functions can be much faster for textual conversion than using the standard FloatToText() implementation. They are validated with provided regression tests.

Of course, in normal code, it is certainly not worth using the Int64 binary representation of currency, but rely on the default compiler/RTL implementation. In all cases, having optimized functions was a need for both speed and accuracy of our ORM data processing, and also for below (page 239).

Note that we discovered some issue in the FPC compiler, when currency is used when compiling from x64-win64: currency values comparison may be wrongly implemented using x87 registers: we found out that using a i386-win32 FPC compiler is a safer approach, even targetting x64-win64 - at least for the trunk in 2019/11.
4.3. TDynArray dynamic array wrapper

Version 1.13 of the SynCommons.pas unit introduced two kinds of wrapper:

- Low-level RTTI functions for handling record types: RecordEquals, RecordSave, RecordSaveLength, RecordLoad;
- TDynArray and TDynArrayHashed objects, which are wrappers around any dynamic array.

With TDynArray, you can access any dynamic array (like TIntegerDynArray = array of integer) using TList-like properties and methods, e.g. Count, Add, Insert, Delete, Clear, IndexOf, Find, Sort and some new methods like LoadFromStream, SaveToStream, LoadFrom, SaveTo, Slice, Reverse, and AddArray. It includes e.g. fast binary serialization of any dynamic array, even containing strings or records - a CreateOrderedIndex method is also available to create individual index according to the dynamic array content. You can also serialize the array content into JSON, if you wish.

One benefit of dynamic arrays is that they are reference-counted, so they do not need any Create/try..finally...Free code, and are well handled by the Delphi compiler. For performance-critical tasks, dynamic array access is very optimized, since its whole content will be allocated at once, therefore reducing the memory fragmentation and being much more CPU cache friendly.

Dynamic arrays are no replacement to a TCollection nor a TList (which are the standard and efficient way of storing class instances, and are also handled as published properties since revision 1.13 of the framework), but they are very handy way of having a list of content or a dictionary at hand, with no previous class nor properties definition.

You can look at them like Python's list, tuples (via records handling) and dictionaries (via Find method, especially with the dedicated TDynArrayHashed wrapper), in pure Delphi. Our new methods (about searching and serialization) allow most usage of those script-level structures in your Delphi code.

In order to handle dynamic arrays in our ORM, some RTTI-based structure were designed for this task. Since dynamic array of records should be necessary, some low-level fast access to the record content, using the common RTTI, has also been implemented (much faster than the "new" enhanced RTTI available since Delphi 2010).

4.3.1. TList-like properties

Here is how you can have method-driven access to the dynamic array:

```delphi
type
  TGroup: array of integer;
var
  Group: TGroup;
  GroupA: TDynArray;
  i, v: integer;
begin
  GroupA.Init(TypeInfo(TGroup),Group); // associate GroupA with Group
  for i := 0 to 1000 do
    begin
      v := i + 1000; // need argument passed as a const variable
      GroupA.Add(v);
    end;
  v := 1500;
  if GroupA.IndexOf(v) < 0 then // search by content
    ShowMessage('Error: 1500 not found!');
  for i := GroupA.Count-1 downto 0 do
```
if \( i \) and \( 3 = 0 \) then
    GroupA.Delete(i); // delete integer at index \( i \)
end;

This TDynArray wrapper will work also with array of string or array of record...

Records need only to be packed and have only not reference counted fields (byte, integer, double...) or string or variant reference-counted fields (there is no support of nested Interface yet). TDynArray is able to handle record within record, and even dynamic arrays within record.

Yes, you read well: it will handle a dynamic array of record, in which you can put some string or whatever data you need.

The IndexOf() method will search by content. That is e.g. for an array of record, all record fields content (including string properties) must match.

Note that TDynArray is just a wrapper around an existing dynamic array variable. In the code above, Add and Delete methods are modifying the content of the Group variable. You can therefore initialize a TDynArray wrapper on need, to access more efficiently any native Delphi dynamic array. TDynArray doesn't contain any data: the elements are stored in the dynamic array variable, not in the TDynArray instance.

### 4.3.2. Enhanced features

Some methods were defined in the TDynArray wrapper, which are not available in a plain TList - with those methods, we come closer to some native generics implementation:
- Now you can save and load a dynamic array content to or from a stream or a string (using LoadFromStream/SaveToStream or LoadFrom/SaveTo methods) - it will use a proprietary but very fast binary stream layout;
- And you can sort the dynamic array content by two means: either in-place (i.e. the array elements content is exchanged - use the Sort method in this case) or via an external integer index look-up array (using the CreateOrderedIndex method - in this case, you can have several orders to the same data);
- You can specify any custom comparison function, and there is a new Find method will can use fast binary search if available.

Here is how those new methods work:

```delphi
var
    Test: RawByteString;
...
    Test := GroupA.SaveTo;
    GroupA.Clear;
    GroupA.LoadFrom(Test);
    GroupA.Compare := SortDynArrayInteger;
    GroupA.Sort;
    for \( i := 1 \) to GroupA.Count-1 do
        if Group[\( i \)]<Group[\( i-1 \)] then
            ShowMessage('Error: unsorted!');
    v := 1500;
    if GroupA.Find(v)<0 \( // \) fast binary search
        ShowMessage('Error: 1500 not found!');
```

Some unique methods like Slice, Reverse or AddArray are also available, and mimic well-known Python methods.

Still closer to the generic paradigm, working for Delphi 6 up to Delphi 10.3 Rio, without the need of the slow enhanced RTTI, nor the executable size overhead and compilation issues of generics...
4.3.3. Capacity handling via an external Count

One common speed issue with the default usage of TDynArray is that the internal memory buffer is reallocated when you change its length, just like a regular Delphi dynamic array.

That is, whenever you call Add or Delete methods, an internal call to SetLength(DynArrayVariable) is performed. This could be slow, because it always executes some extra code, including a call to ReallocMem.

In order not to suffer for this, you can define an external Count value, as an Integer variable.

In this case, the Length(DynArrayVariable) will be the memory capacity of the dynamic array, and the exact number of stored item will be available from this Count variable. A Count property is exposed by TDynArray, and will always reflect the number of items stored in the dynamic array. It will point either to the external Count variable, if defined; or it will reflect the Length(DynArrayVariable), just as usual. A Capacity property is also exposed by TDynArray, and will reflect the capacity of the dynamic array: in case of an external Count variable, it will reflect Length(DynArrayVariable).

As a result, adding or deleting items could be much faster.

```delphi
var
  Group: TIntegerDynArray;
  GroupA: TDynArray;
  GroupCount, i, v: integer;
begin
  GroupA.Init(TypeInfo(TGroup), Group, @GroupCount);
  GroupA.Capacity := 1023; // reserve memory
  for i := 0 to 1000 do
    begin
      v := i + 1000; // need argument passed as a const variable
      GroupA.Add(v); // faster than with no external GroupCount variable
    end;
  Check(GroupA.Count = 1001);
  Check(GroupA.Capacity = 1023);
  Check(GroupA.Capacity = length(Group));
end;
```

4.3.4. JSON serialization

The TDynArray wrapper features some native JSON serialization abilities: TTextWriter, AddDynArrayJSON and TDynArray. LoadFromJSON methods are available for UTF-8 JSON serialization of dynamic arrays.

See below (page 303) for all details about this unique feature.

4.3.5. Daily use

The TTestLowLevelCommon._TDynArray and _TDynArrayHashed methods implement the automated unitary tests associated with these wrappers.

You'll find out there samples of dynamic array handling and more advanced features, with various kind of data (from plain TIntegeryDynArray to records within records).

The TDynArrayHashed wrapper allow implementation of a dictionary using a dynamic array of record. For instance, the prepared statement cache is handling by the following code in SynSQLite3.pas:

```delphi
TSQLStatementCache = record
  StatementSQL: RawUTF8;
  Statement: TSQLRequest;
end;
```
Those definitions will prepare a dynamic array storing a TSQLRequest and SQL statement association, with an external Count variable, for better speed.

It will be used as such in TSQLRestServerDB:

```
constructor TSQLRestServerDB.Create(aModel: TSQLModel; aDB: TSQLDataBase);
begin
  fStatementCache.Init(aDB);
  (...)
end;
```

The wrapper will be initialized in the object constructor:

```
procedure TSQLStatementCached.Init(aDB: TSQLite3DB);
begin
  Caches.Init(TypeInfo(TSQLStatementCacheDynArray),Cache,nil,nil,nil,@Count);
  DB := aDB;
end;
```

The TDynArrayHashed.Init method will recognize that the first TSQLStatementCache field is a RawUTF8, so will set by default an AnsiString hashing of this first field (we could specify a custom hash function or content hashing by overriding the default nil parameters to some custom functions).

So we can specify directly a GenericSQL variable as the first parameter of FindHashedForAdding, since this method will only access to the first field RawUTF8 content, and won't handle the whole record content. In fact, the FindHashedForAdding method will be used to make all the hashing, search, and new item adding if necessary - just in one step. Note that this method only prepare for adding, and code needs to explicitly set the StatementSQL content in case of an item creation:

```
function TSQLStatementCached.Prepare(const GenericSQL: RawUTF8): PSQLRequest;
var
  added: boolean;
begin
  with Cache[Caches.FindHashedForAdding(GenericSQL, added)] do begin
    if added then begin
      StatementSQL := GenericSQL; // need explicit set the content
      Statement.Prepare(DB, GenericSQL);
    end else begin
      Statement.Reset;
      Statement.BindReset;
    end;
    result := @Statement;
  end;
end;
```

The latest method of TSQLStatementCached will just loop for each statement, and close them: you can note that this code uses the dynamic array just as usual:

```
procedure TSQLStatementCached.ReleaseAllDBStatements;
var
  i: integer;
begin
  for i := 0 to Count-1 do
    Cache[i].Statement.Close; // close prepared statement
  Caches.Clear; // same as SetLength(Cache,0) + Count := 0
end;
```
The resulting code is definitively quick to execute, and easy to read/maintain.

### 4.3.6. TDynArrayHashed

If your purpose is to access a dynamic array using one of its fields as key, consider using TDynArrayHashed. This wrapper, inheriting from TDynArray, will store an hashed index of one field of the dynamic array record, for very efficient lookup. For a few dozen entries, it won't change the performance, but once you reach thousands of items, an index will be much faster - almost O(1) instead of O(n).

In respect to TDynArray, TDynArrayHashed instance lifetime should be consistent with the dynamic array itself, to ensure the hashed index is properly populated. You should also ensure that the dynamic array content is modified mainly via the TDynArrayHashed.FindHashedForAdding TDynArrayHashed.FindHashedAndUpdate and TDynArrayHashed.FindHashedAndDelete methods, or explicitly call TDynArrayHashed.ReHash when the dynamic array content has been modified.

In practice, TDynArrayHashed.FindHashed will be much faster than a regular TDynArray.Find call.

### 4.3.7. TSynDictionary

One step further is available with the TSynDictionary class. It is a thread-safe dictionary to store some values from associated keys, as two separated dynamic arrays.

Each TSynDictionary instance will hold and store the associated dynamic arrays - this is not the case with TDynArray and TDynArrayHashed, which are only wrappers around an existing dynamic array variable.

One big advantage is that access to TSynDictionary methods are thread-safe by design: internally, a TSynLock will protect the keys, maintained by a TDynArrayHashed (page 111) instance, and the values by a TDynArray. Access to/from local variables will be made via explicit copy, for perfect thread safety.

For advanced use, the TSynDictionary offers JSON serialization and binary storage (with optional compression), and the ability to specify a timeout period in seconds, after which any call to TSynDictionary.DeleteDeprecated will delete older entries - which is very convenient to cache values, with optional persistence on disk. Just like your own in-process Redis/MemCached instance.
4.4. TDocVariant custom variant type

With revision 1.18 of the framework, we introduced two new custom types of variants:
- TDocVariant kind of variant;
- TBSONVariant kind of variant.

The second custom type (which handles MongoDB-specific extensions - like ObjectId or other specific types like dates or binary) will be presented later, when dealing with MongoDB support in mORMot, together with the BSON kind of content. BSON / MongoDB support is implemented in the SynMongoDB.pas unit.

We will now focus on TDocVariant itself, which is a generic container of JSON-like objects or arrays. This custom variant type is implemented in SynCommons.pas unit, so is ready to be used everywhere in your code, even without any link to the mORMot ORM kernel, or MongoDB.

4.4.1. TDocVariant documents

TDocVariant implements a custom variant type which can be used to store any JSON/BSON document-based content, i.e. either:
- Name/value pairs, for object-oriented documents (internally identified as dvObject sub-type);
- An array of values (including nested documents), for array-oriented documents (internally identified as dvArray sub-type);
- Any combination of the two, by nesting TDocVariant instances.

Here are the main features of this custom variant type:
- DOM approach of any object or array documents;
- Perfect storage for dynamic value-objects content, with a schema-less approach (as you may be used to in scripting languages like Python or JavaScript);
- Allow nested documents, with no depth limitation but the available memory;
- Assignment can be either per-value (default, safest but slower when containing a lot of nested data), or per-reference (immediate reference-counted assignment);
- Very fast JSON serialization / un-serialization with support of MongoDB-like extended syntax;
- Access to properties in code, via late-binding (including almost no speed penalty due to our VCL hack as detailed in SDD # DI-2.2.3);
- Direct access to the internal variant names and values arrays from code, by trans-typing into a TDocVariantData record;
- Instance life-time is managed by the compiler (like any other variant type), without the need to use interfaces or explicit try..finally blocks;
- Optimized to use as little memory and CPU resource as possible (in contrast to most other libraries, it does not allocate one class instance per node, but rely on pre-allocated arrays);
- Opened to extension of any content storage - for instance, it will perfectly integrate with BSON serialization and custom MongoDB types (ObjectId, Decimal128, RegEx...), to be used in conjunction with MongoDB servers;
- Perfectly integrated with our TDynArray dynamic array wrapper (page 107) and its JSON serialization - see below (page 303), as with the record serialization - see below (page 297);
- Designed to work with our mORMot ORM: any TSQLRecord instance containing such variant custom types as published properties will be recognized by the ORM core, and work as expected with any database back-end (storing the content as JSON in a TEXT column);
- Designed to work with our mORMot SOA: any interface-based service - see below (page 419) - is able to consume or publish such kind of content, as variant kind of parameters;
- Fully integrated with the *Delphi* IDE: any variant instance will be displayed as JSON in the IDE debugger, making it very convenient to work with.

To create instances of such variant, you can use some easy-to-remember functions:
- `_Obj() _ObjFast()` global functions to create a variant *object* document;
- `_Arr() _ArrFast()` global functions to create a variant *array* document;
- `_Json() _JsonFast() _JsonFmt() _JsonFastFmt()` global functions to create any variant *object* or *array* document from JSON, supplied either with standard or *MongoDB*-extended syntax.

You have two non-excluding ways of using the TDocVariant storage:
- As regular variant variables, then using either late-binding or faster `_Safe()` to access its data;
- Directly as TDocVariantData variables, then later on returing a variant instance using `variant(aDocVariantData)`.

Note that you do not need to protect any stack-allocated TDocVariantData instance with a `try..finally`, since the compiler will do it for you. This record type has a lot of powerful methods, e.g. to apply `map/reduce` on the content, or do advanced searches or marshalling.

### 4.4.1.1. Variant object documents

The more straightforward is to use late-binding to set the properties of a new TDocVariant instance:

```delphi
var V: variant;
...
TDocVariant.New(V); // or slightly slower V := TDocVariant.New;
V.name := 'John';
V.year := 1972;
// now V contains{"name":"john","year":1982}
```

With `_Obj()`, an *object* variant instance will be initialized with data supplied two by two, as *Name,Value* pairs, e.g.

```delphi
var V1,V2: variant; // stored as any variant
...
V1 := _Obj([\'name\',\'John\',\'year\',1972]);
V2 := _Obj([\'name\',\'John\',\'doc\',_Obj([\'one\',1,\'two\',2.5])]); // with nested objects
```

Then you can convert those objects into JSON, by two means:
- Using the `VariantSaveJson()` function, which return directly one UTF-8 content;
- Or by trans-typing the variant instance into a string (this will be slower, but is possible).

```delphi
writeln(VariantSaveJson(V1)); // explicit conversion into RawUTF8
writeln(V1); // implicit conversion from variant into string
// both commands will write '{"name":"john","year":1982}'
writeln(VariantSaveJson(V2)); // explicit conversion into RawUTF8
writeln(V2); // implicit conversion from variant into string
// both commands will write '{"name":"john","doc":{"one":1,"two":2.5}}
```

As a consequence, the *Delphi* IDE debugger is able to display such variant values as their JSON representation. That is, V1 will be displayed as `{"name":"john","year":1982}` in the IDE debugger *Watch List* window, or in the *Evaluate/Modify* (F7) expression tool. This is pretty convenient, and much more user-friendly than any class-based solution (which requires the installation of a specific design-time package in the IDE).

You can access to the object properties via late-binding, with any depth of nesting objects, in your code:

```delphi
writeln('name=',V1.name,' year=',V1.year);
// will write 'name=John year=1972'
writeln('name=',V2.name,' doc.one=',V2.doc.one,' doc.two=',V2.doc.two);
```
// will write 'name=John doc.one=1 doc.two=2.5
V1.name := 'Mark';  // overwrite a property value
writeln(V1.name);  // will write 'Mark'
V1.age := 12;     // add a property to the object
writeln(V1.age);  // will write '12'

Note that the property names will be evaluated at runtime only, not at compile time. For instance, if you write V1.nome instead of V1.name, there will be no error at compilation, but an EDocVariant exception will be raised at execution (unless you set the dvoReturnNullForUnknownProperty option to _Obj/_Arr/_Json/_JsonFmt which will return a null variant for such undefined properties).

In addition to the property names, some pseudo-methods are available for such object variant instances:

writeln(V1._Count);  // will write 3 i.e. the number of name/value pairs in the object document
writeln(V1._Kind);   // will write 1 i.e. ord(dvObject)
for i := 0 to V2._Count-1 do
  writeln(V2.Name(i), '=', V2.Value(i));
// will write to the console:
// name=John
//   doc={"one":1,"two":2.5}
//   age=12
if V1.Exists('year') then
  writeln(V1.year);
V1.Add('key', 'value');   // add one property to the object

The variant values returned by late-binding are generated as varByRef, so it has two benefits:
- Much better performance, even if the nested objects are created per-value (see below);
- Allow nested calls of pseudo methods, as such:

var V: variant;
...
V := _Json('{arr: [1,2]}');
V.arr.Add(3);    // will work, since V.arr is returned by reference (varByRef)
writeln(V);     // will write '{"arr": [1,2,3]}'
V.arr.Delete(1);
writeln(V);     // will write '{"arr": [1,3]}'

You may also trans-type your variant instance into a TDocVariantData record, and access directly to its internals.
For instance:

TDocVariantData(V1).AddValue('comment', 'Nice guy');
with TDocVariantData(V1) do // direct trans-typing
  if Kind=dvObject then    // direct access to the TDocVariantKind field
    for i := 0 to Count-1 do  // direct access to the Count: integer field
      writeln(Names[i], '=', Values[i]); // direct access to the internal storage arrays

By definition, trans-typing via a TDocVariantData record is slightly faster than using late-binding.

But you must ensure that the variant instance is really a TDocVariant kind of data before trans-typing e.g. by calling _Safe(aVariant)^ function (or DocVariantType.IsOfType(aVariant) or DocVariantData(aVariant)^), which will work even for members returned as varByRef via late binding (e.g. V2.doc):

with _Safe(V1)^ do  // note ^ to de-reference into TDocVariantData
  for ndx := 0 to Count-1 do  // direct access to the Count: integer field
    writeln(Names[ndx], '=', Values[ndx]);  // direct access to the internal storage arrays

writeln(V2.doc);  // will write '{"name":"john","doc":{"one":1,"two":2.5}}'
if DocVariantType.IsOfType(V2.Doc) then  // will be false, since V2.Doc is a varByRef variant
  writeln('never run');  // so TDocVariantData(V2.doc) will fail
with DocVariantData(V2.Doc)^ do  // note ^ to de-reference into TDocVariantData
  for ndx := 0 to Count-1 do  // direct access the TDocVariantData methods
    writeln(Names[ndx], '=', Values[ndx]);
In practice, `_Safe(aVariant)` may be preferred, since `DocVariantData(aVariant)` will raise an EDocVariant exception if `aVariant` is not a `TDocVariant`, but `_Safe(aVariant)` will return a "fake" void `DocVariant` instance, in which Count=0 and Kind=dbUndefined.

The `TDocVariantData` type features some additional `U[]` `I[]` `B[]` `D[]` `O[]` `A[]` `A_[]` `_[]` properties, which could be used to have direct typed access to the data, as RawUTF8, Int64/integer, Double, or checking if the nested document is an `O[]`bject or an `A[]`rray.

You can also allocate directly the `TDocVariantData` instance on stack, if you do not need any variant-oriented access to the object, but just some local storage:

```pascal
var Doc1, Doc2: TDocVariantData;
...
Doc1.Init; // needed for proper initialization
assert(Doc1.Kind=dvUndefined);
Doc1.AddValue('name','John');    // add some properties
Doc1.AddValue('birthyear',1972);
assert(Doc1.Kind=dvObject);     // is now identified as an object
assert(Doc1.Value['name']=='John'); // read access to the properties (also as varByRef)
assert(Doc1.Value['birthyear']==1972);
assert(Doc1.U['name']=='John'); // slightly faster read access
assert(Doc1.U['birthyear']==1972);
writeln(Doc1.ToJSON); // will write '{"name":"John","birthyear":1972}"
Doc1.Value['name'] := 'Jonas';  // update one property
writeln(Doc1.ToJSON); // will write '{"name":"Jonas","birthyear":1972}"
Doc1.InitObject(['name','John','birthyear',1972],
    aOptions+[dvoReturnNullForUnknownProperty]); // initialization from name/value pairs
assert(Doc2.Kind=dvObject);
assert(Doc2.Count=2);
assert(Doc2.Names[0]=='name');
assert(Doc2.Values[0]=='John');
writeln(Doc2.ToJSON); // will write '{"name":"John","birthyear":1972}"
Doc2.Delete('name');
writeln(Doc2.ToJSON); // will write '{"birthyear":1972}"
assert(Doc2.U['name']='');
assert(Doc2.U['birthyear']==1972);
Doc2.U['name'] := 'Paul';
Doc2.U['birthyear'] := 1982;
writeln(Doc2.ToJSON); // will write '{"name":"Paul","birthyear":1982}"
```

You do not need to protect the stack-allocated `TDocVariantData` instances with a `try..finally`, since the compiler will do it for your. Take a look at all the methods and properties of `TDocVariantData`.

### 4.4.1.2. FPC restrictions

You should take note that with the FreePascal compiler, calling late-binding functions with arguments (like Add or Delete) would most probably fail to work as expected.

We have found out that the following code may trigger some random access violations:

```pascal
doc.Add('text');
doc.Add(anotherdocvariant);
```

So you should rather access directly the underlying `TDocVariantData` instance:

```pascal
TDocVariantData(doc).AddItem('text');
TDocVariantData(doc).AddItem(anotherdocvariant);
```

Or even better using `_Safe()`:

```pascal
_Safe(doc)^.AddItem('text');
```
In fact, late-binding functions arguments seem to work only for simple values (like integer or double), but not complex types (like string or other TDocVariantData), which generate some random GPF, especially when heaptrc paranoid memory checks are enabled.

As a result, direct access to TJSONVariantData instances - preferably via _Safe(), and not a variant variable, will be faster and less error-prone when using FPC.

4.4.1.3. Variant array documents

With _Arr(), an array variant instance will be initialized with data supplied as a list of Value1,Value2,..., e.g.

```delphi
var V1,V2: variant; // stored as any variant
...
V1 := _Arr(['John','Mark','Luke']);
V2 := _Obj(['name','John','array',_Arr(['one','two',2.5])]); // as nested array
```

Then you can convert those objects into JSON, by two means:
- Using the VariantSaveJson() function, which return directly one UTF-8 content;
- Or by trans-typing the variant instance into a string (this will be slower, but is possible).

```delphi
writeln(VariantSaveJson(V1));
// implicit conversion from variant into string
// both commands will write ['"John","Mark","Luke"']
writeln(VariantSaveJson(V2));
// implicit conversion from variant into string
// both commands will write '{"name":"John","array:['"one","two",2.5"]}'
```

As a with any object document, the Delphi IDE debugger is able to display such array variant values as their JSON representation.

Late-binding is also available, with a special set of pseudo-methods:

```delphi
writeln(V1._Count); // will write 3 i.e. the number of items in the array document
writeln(V1._Kind); // will write 2 i.e. ord(dvArray)
for i := 0 to V1._Count-1 do
  writeln(V1.Value(i),':',V2._(i)); // Value() or _() pseudo-methods
  // will write in the console:
  //  John John
  //  Mark Mark
if V1.Exists('John') then
  writeln('John found in array');
V1.Add('new item'); // add "new item" to the array
V1._ := 'another new item'; // add "another new item" to the array
writeln(V1); // will write ['"John","Mark","Luke","new item","another new item"']
V1.Delete(2);
V1.Delete(1);
writeln(V1); // will write ['"John","Luke","another new item"']
```

When using late-binding, the object properties or array items are retrieved as varByRef, so you can even run the pseudo-methods on any nested member:

```delphi
V := _Json('["root","name":"Jim","year":1972]');
V.Add(3.1415);
assert(V['root','name':'Jim','year':1972],3.1415); // delete a property of the nested object
assert(V['root','name':'Jim'],3.1415);
V.Delete(1); // delete an item in the main array
assert(V['root',3.1415]);
```

Of course, trans-typing into a TDocVariantData record is possible, and will be slightly faster than using late-binding. As usual, using _Safe(aVariant)^ function is safer, especially when working on
Synopse mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

varByRef members returned via late-binding.

As with an object document, you can also allocate directly the TDocVariantData instance on stack, if
you do not need any variant-oriented access to the array:
var Doc: TDocVariantData;
...
Doc.Init; // needed for proper initialization - see also Doc.InitArray()
assert(Doc.Kind=dvUndefined);
// this instance has no defined sub-type
Doc.AddItem('one');
// add some items to the array
Doc.AddItem(2);
assert(Doc.Kind=dvArray);
// is now identified as an array
assert(Doc.Value[0]='one');
// direct read access to the items
assert(Doc.Values[0]='one');
// with index check
assert(Doc.Count=2);
writeln(Doc.ToJSON); // will write '["one",2]'
Doc.Delete(0);
assert(Doc.Count=1);
writeln(Doc.ToJSON); // will write '[2]'

You could use the A[] property to retrieve an object property as a TDocVariant array, or the A_[]
property to add a missing array property to an object, for instance:
Doc.Clear; // reset the previous Doc
writeln(Doc.A['test']); // will write
Doc.A_['test']^.AddItems([1,2]);
writeln(Doc.ToJSON);
// will write
writeln(Doc.A['test']); // will write
Doc.A_['test']^.AddItems([3,4]);
writeln(Doc.ToJSON);
// will write

content
'null'
'{"test":[1,2]}'
'[1,2]'
'{"test":[1,2,3,4]}'

4.4.1.4. Create variant object or array documents from JSON
With _Json() or _JsonFmt(), either a document or array variant instance will be initialized with
data supplied as JSON, e.g.
var V1,V2,V3,V4: variant; // stored as any variant
...
V1 := _Json('{"name":"john","year":1982}'); // strict JSON syntax
V2 := _Json('{name:"john",year:1982}');
// with MongoDB extended syntax for names
V3 := _Json('{"name":?,"year":?}',[],['john',1982]);
V4 := _JsonFmt('{%:?,%:?}',['name','year'],['john',1982]);
writeln(VariantSaveJSON(V1));
writeln(VariantSaveJSON(V2));
writeln(VariantSaveJSON(V3));
// all commands will write '{"name":"john","year":1982}'

Of course, you can nest objects or arrays as parameters to the _JsonFmt() function.
The supplied JSON can be either in strict JSON syntax, or with the MongoDB extended syntax, i.e. with
unquoted property names. It could be pretty convenient and also less error-prone when typing in the
Delphi code to forget about quotes around the property names of your JSON.
Note that TDocVariant implements an open interface for adding any custom extensions to JSON: for
instance, if the SynMongoDB.pas unit is defined in your application, you will be able to create any
MongoDB specific types in your JSON, like ObjectID(), NumberDecimal(""...") ,new Date() or even
/regex/option.
As a with any object or array document, the Delphi IDE debugger is able to display such variant
values as their JSON representation.
4.4.1.5. Per-value or per-reference
SAD - mORMot Framework - Rev. 1.18

Page 118 of 2552


By default, the variant instance created by _Obj() _Arr() _Json() _JsonFmt() will use a **copy-by-value** pattern. It means that when an instance is affected to another variable, a new variant document will be created, and all internal values will be copied. Just like a record type.

This will imply that if you modify any item of the copied variable, it won't change the original variable:

```delphi
var V1,V2: variant;
...
V1 := _Obj(['name','John','year',1972]);
V2 := V1;
// create a new variant, and copy all values
v2.name := 'James';  // modifies V2.name, but not V1.name
writeln(V1.name, ' and ',V2.name);
// will write 'John and James'
```

As a result, your code will be perfectly safe to work with, since V1 and V2 will be uncoupled.

But one drawback is that passing such a value may be pretty slow, for instance, when you nest objects:

```delphi
var V1,V2: variant;
...
V1 := _Obj(['name','John','year',1972]);
V2 := _Arr(['John','Mark','Luke']);
V1.names := V2;  // here the whole V2 array will be re-allocated into V1.names
```

Such a behavior could be pretty time and resource consuming, in case of a huge document.

All _Obj() _Arr() _Json() _JsonFmt() functions have an optional TDocVariantOptions parameter, which allows to change the behavior of the created TDocVariant instance, especially setting dvoValueCopiedByReference.

This particular option will set the **copy-by-reference** pattern:

```delphi
var V1,V2: variant;
...
V1 := _Obj(['name','John','year',1972],[dvoValueCopiedByReference]);
V2 := V1;
// creates a reference to the V1 instance
V2.name := 'James';  // modifies V2.name, but also V1.name
writeln(V1.name, ' and ',V2.name);
// will write 'James and James'
```

You may think that this behavior is somewhat weird for a variant type. But if you forget about **per-value** objects and consider those TDocVariant types as a Delphi class instance (which is a **per-reference** type), without the need of having a fixed schema nor handling manually the memory, it will probably start to make sense.

Note that a set of global functions have been defined, which allows direct creation of documents with **per-reference** instance lifetime, named _ObjFast() _ArrFast() _JsonFast() _JsonFmtFast(). Those are just wrappers around the corresponding _Obj() _Arr() _Json() _JsonFmt() functions, with the following JSON_OPTIONS[true] constant passed as options parameter:

```delphi
const
/// some convenient TDocVariant options
/// - JSON_OPTIONS[false] is _Json() and _JsonFmt() functions default
/// - JSON_OPTIONS[true] are used by _JsonFast() and _JsonFastFmt() functions
JSON_OPTIONS: array[Boolean] of TDocVariantOptions = (  
[dvoReturnNullForUnknownProperty],
[dvoReturnNullForUnknownProperty,dvoValueCopiedByReference]);
```

When working with complex documents, e.g. with BSON / MongoDB documents, almost all content will be created in "fast" **per-reference** mode.

### 4.4.2. Advanced TDocVariant process
4.4.2.1. Number values options

By default, TDocVariantData will only recognize integer, Int64 and currency - see Currency handling (page 106) - as number values. Any floating point value which may not be translated to/from JSON textual representation safely will be stored as a JSON string, i.e. if it does match an integer or up to 4 fixed decimals, with 64-bit precision. We stated that JSON serialization should be conservative, i.e. serializing then unserializing (or the other way round) should return the very same value; parsing JSON is a matter of (difficult) choices - see http://seriot.ch/parsing_json.php#5.. - and we choose to be paranoid and not loose information by default.

You can set the dvoAllowDoubleValue option to TDocVariantData, so that such floating-point numbers will be recognized and stored as double. In this case, only varDouble storage will be used for the variant values, i.e. 64-bit IEEE 754 double values, handling 5.0 x 10^-324 .. 1.7 x 10^308 range. With such floating-point values, you may loose precision and digits during the JSON serialization process: this is why it is not enabled by default.

Also note that some JSON engines do not support 64-bit integer numbers. For instance, JavaScript engines handle only up to 53-bit of information without precision loss (called the significand bits), due to their internal storage as a 8 bytes IEEE 754 container. In some cases, it is safest to use JSON string representation of such numbers, as is done with the woIDAsIDstr value of TTextWriterWriteObjectOption for safe serialization of TSQLRecord.ID ORM values.

If you want to work with high-precision floating point numbers, consider using TDecimal128 values, as implemented in SynMongoDB.pas, which supports 128-bit high precision decimal, as defined by the IEEE 754-2008 128-bit decimal floating point standard, and handled in MongoDB 3.4+. Their conversion to/from text - therefore to/from JSON - won't loose nor round any digit, as soon as the value fits in its 128-bit storage.

4.4.2.2. Object or array document creation options

As stated above, a TDocVariantOptions parameter enables to define the behavior of a TDocVariant custom type for a given instance. Please refer to the documentation of this set of options to find out the available settings. Some are related to the memory model, other to case-sensitivity of the property names, other to the behavior expected in case of non-existing property, and so on...

Note that this setting is local to the given variant instance.

In fact, TDocVariant does not force you to stick to one memory model nor a set of global options, but you can use the best pattern depending on your exact process. You can even mix the options - i.e. including some objects as properties in an object created with other options - but in this case, the initial options of the nested object will remain. So you should better use this feature with caution.

You can use the _Unique() global function to force a variant instance to have an unique set of options, and all nested documents to become by-value, or _UniqueFast() for all nested documents to become by-reference.

```plaintext
// assuming V1='{"name":"James","year":1972}' created by-reference
_unique(V1);  // change options of V1 to be by-value
V2 := V1;      // creates a full copy of the V1 instance
V2.name := 'John'; // modifies V2.name, but not V1.name
writeln(V1.name); // write 'James'
writeln(V2.name); // write 'John'
V1 := _Arr(['root',V2]); // created as by-value by default, as V2 was
writeln(V1._Count); // write 2
_uniqueFast(V1);  // change options of V1 to be by-reference
V2 := V1;
V1.(1).name := 'Jim';
```
The easiest is to stick to one set of options in your code, i.e.:

- Either using the _*() global functions if your business code does send some TDocVariant instances to any other part of your logic, for further storage: in this case, the *by-value* pattern does make sense;
- Or using the _*Fast() global functions if the TDocVariant instances are local to a small part of your code, e.g. used as dynamic schema-less *Data Transfer Objects (DTO)*.

In all cases, be aware that, like any class type, the const, var and out specifiers of method parameters does not behave to the TDocVariant value, but to its reference.

### 4.4.2.3. Integration with other mORMot units

In fact, whenever a dynamic schema-less storage structure is needed, you may use a TDocVariant instance instead of class or record strong-typed types:

- Client-Server ORM - see below (page 130) - will support TDocVariant in any of the TSQLRecord variant published properties (and store them as JSON in a text column);
- Interface-based services - see below (page 419) - will support TDocVariant as variant parameters of any method, which make them as perfect DTO;
- Since JSON support is implemented with any TDocVariant value from the ground up, it makes a perfect fit for working with AJAX clients, in a script-like approach;
- If you use our SynMongoDB.pas mORMotMongoDB.pas units to access a MongoDB server, TDocVariant will be the native storage to create or access nested BSON arrays or objects documents - that is, it will allow proper ODM storage;
- Cross-cutting features (like logging or record / dynamic array enhancements) will also benefit from this TDocVariant custom type.

We are pretty convinced that when you will start playing with TDocVariant, you won't be able to live without it any more. It introduces the full power of late-binding and dynamic schema-less patterns to your application code, which can be pretty useful for prototyping or in Agile development. You do not need to use scripting engines like Python or JavaScript: Delphi is perfectly able to handle dynamic coding!

### 4.5. Cross-cutting functions

#### 4.5.1. Iso8601 time and date

For date/time storage as text, the framework will use *ISO 8601* encoding. Dates could be encoded as YYYY-MM-DD or YYYYMMDD, time as hh:mm:ss or hhmmss, and combined date and time representations as <date>T<time>, i.e. YYYY-MM-DDThh:mm:ss or YYYYMMDDThhmmss.

The *lexicographical order* of the representation thus corresponds to chronological order, except for date representations involving negative years. This allows dates to be naturally sorted by, for example, file systems, or grid lists.

#### 4.5.1.1. TDateTime and TDateTimeMS

In addition to the default TDateTime type, which will be serialized with a second resolution, you may use TDateTimeMS, which will include the milliseconds, i.e. YYYY-MM-DDThh:mm:ss.sss or YYYYMMDDThhmmss.sss.
**4.5.1.2. TTTimeLog and TTTimeLogBits**

The SynCommons.pas unit defines a TTTimeLog type, and some functions able to convert to/from regular TDateTime values:

```pascal
type
  TTTimeLog = type Int64;
```

This integer storage is encoded as a series of bits, which will map the TTTimeLogBits record type, as defined in SynCommons.pas unit.

The resolution of such values is one second. In fact, it uses internally for computation an abstract "year" of 16 months of 32 days of 32 hours of 64 minutes of 64 seconds.

As a consequence, any date/time information can be retrieved from its internal bit-level representation:
- 0..5 bits will map *seconds*,
- 6..11 bits will map *minutes*,
- 12..16 bits will map *hours*,
- 17..21 bits will map *days* (minus one),
- 22..25 bits will map *months* (minus one),
- 26..40 bits will map *years*.

The ISO 8601 standard allows millisecond resolution, encoded as hh:mm:ss.sss or hhmmss.sss. Our TTTimeLog/TTTimeLogBits integer encoding uses a second time resolution, and a 64-bit integer storage, so is not able to handle such precision. You could use TDateTimeMS or TUnixMSTime values instead, if milliseconds are required.

Note that since TTTimeLog type is bit-oriented, you can't just use *add* or *subtract* two TTTimeLog values when doing such date/time computation: use a TDateTime temporary conversion in such case. See for instance how the TSQLRest.ServerTimestamp property is computed:

```pascal
function TSQLRest.GetServerTimestamp: TTTimeLog;
begin
  PTimeLogBits(@result)^.From(Now+fServerTimestampOffset);
end;
```

```pascal
procedure TSQLRest.SetServerTimestamp(const Value: TTTimeLog);
begin
  fServerTimestampOffset := PTimeLogBits(Value)^.ToDateTime-Now;
end;
```

But if you simply want to *compare* TTTimeLog kind of date/time, it is safe to directly compare their Int64 underlying value, since timestamps will be stored in increasing order, with a resolution of one second.

Due to compiler limitation in older versions of Delphi, direct typecast of a TTTimeLog or Int64 variable into a TTTimeLogBits record (as with TTTimeLogBits(aTimeLog).ToDateTime) could lead to an internal compiler error. In order to circumvent this bug, you will have to use a pointer typecast, e.g. as in TTTimeLogBits(@Value)^.ToDateTime above.

But in most case, you should better use the following functions to manage such timestamps:

```pascal
function TTLogNow: TTTimeLog;
function TTLogNowUTC: TTTimeLog;
```
function TimeLogFromDateTime(DateTime: TDateTime): TTimeLog;
function TimeLogToDateTime(const Timestamp: TTimeLog): TDateTime; overload;
function Iso8601ToTimeLog(const S: RawByteString): TTimeLog;

See below (page 136) for additional information about this TTimeLog storage, and how it is handled by the framework ORM, via the additional TModTime and TCreateTime types.

### 4.5.1.3. TUnixTime and TUnixMSTime

You may consider the TUnixTime type, which holds a 64-bit encoded number of *seconds* since the Unix Epoch, i.e. 1970-01-01 00:00:00 UTC:

```plaintext
type TUnixTime = type Int64;
```

You can convert such values:
- to/from TTimeLog values using TTimeLogBits.ToUnixTime and TTimeLogBits.FromUnixTime methods;
- to/from TDateTime values using UnixTimeToDateTime/DateTimeToUnixTime functions;
- using UnixTimeUTC to return the current timestamp, calling very fast OS API.

An alternative TUnixMSTime type is also available, which stores the date/time as a 64-bit encoded number of *milliseconds* since the Unix Epoch, i.e. 1970-01-01 00:00:00 UTC. Milliseconds resolution may be handy in some cases, especially when TTimeLog second resolution is not enough, and you want a more standard encoding than Delphi's TDateTime.

You may consider using TUnixTime and TUnixMSTime especially if the timestamp is likely to be handled by third-party clients following this C/C#/Java/JavaScript encoding. In the Delphi world, TDateTime, TDateTimeMS or TTimeLog types could be preferred.

### 4.5.2. Time Zones

One common problem when handling dates and times, is that common time is shown and entered as *local*, whereas the computer should better use non-geographic information - especially on a Client-Server architecture, where both ends may not be on the same physical region.

A *time zone* is a region that observes a uniform standard time for legal, commercial, and social purposes. Time zones tend to follow the boundaries of countries and their subdivisions because it is convenient for areas in close commercial or other communication to keep the same time. Most of the time zones on land are offset from *Coordinated Universal Time* (UTC) by a whole number of hours, or minutes. Even worse, some countries use daylight saving time for part of the year, typically by changing clocks by an hour, twice every year.

The main rule is that any date and time stored should be stored in UTC, or with an explicit Zone identifier (i.e. an explicit offset to the UTC value). Our framework expects this behavior: every date/time value stored and handled by the ORM, SOA, or any other part of it, is expected to be UTC-encoded. At presentation layer (e.g. the User Interface), conversion to/from local times should take place, so that the end-user is provided with friendly clock-wall compatible timing.

As you may guess, handling time zones is a complex task, which should be managed by the Operating System itself. Since this cultural material is constantly involving, it is updated as part of the OS.

In practice, current local time could be converted from UTC from the current system-wide time zone. One of the only parameters you have to set when installing an Operating System is to pickup the keyboard layout... and the current time zone to be used. But in a client-server environment, you may have to manage several time zones on the server side: so you can’t rely on this global setting.
One sad - but predictable - disappointment is that there is no common way of encoding time zone information. Under Windows, the registry contains a list of time zones, and the associated time bias data. Most POSIX systems (including Linux and Mac OSX) do rely on the IANA database, also called tzdata - you may have noticed that this particular package is often updated with your system. Both zone identifiers do not map, so our framework needed something to be shared on all systems.

The SynCommons.pas unit features the TSynTimeZone class, which is able to retrieve the information from the Windows registry into memory via TSynTimeZone.LoadFromRegistry, or into a compressed file via TSynTimeZone.SaveToFile. Later on, this file could be reloaded on any system, including any Linux flavor, via TSynTimeZone.LoadFromFile, and returns the very same results. The compressed file is pretty small, thanks to its optimized layout, and use of our SynLZ compression algorithm: the full information is stored in a 7 KB file - the same flattened information as JSON is around 130 KB, and you may compare with the official http://www.iana.org.. content, which weighted as a 280KB tar.gz... Of course, tzdata stores potentially a lot more information than we need.

In practice, you may use TSynTimeZone.Default, which will return an instance read from the current version of the registry under Windows, and will attempt to load the information named after the executable file name (appended as a .tz extension) on other Operating Systems. You may therefore write:

```pascal
aLocalTime := TSynTimeZone.Default.NowToLocal(aTimeZoneID);
```

Similarly, you may use TSynTimeZone.UtcToLocal or TSynTimeZone.LocalToUtc methods, with the proper TZ identifier.

You will have to create the needed .tz compressed file under a Windows machine, then provide this file together with any Linux server executable, in its very same folder. On a Cloud-like system, you may store this information in a centralized server, e.g. via a dedicated service - see below (page 419) - generated from a single reference Windows system via TSynTimeZone.SaveToBuffer, and later on use TSynTimeZone.LoadFromBuffer to decode it from all your cloud nodes. The main benefit is that the time information will stay consistent whatever system it runs on, as you may expect.

Your User Interface could retrieve the IDs and ready to be displayed text from TSynTimeZone.Ids and TSynTimeZone.Displays properties, as plain TStrings instance, which index will follow the TSynTimeZone.Zone[] internal information.

As a nice side effect, the TSynTimeZone binary internal storage has been found out to be very efficient, and much faster than a manual reading of the Windows registry. Complex local time calculation could be done on the server side, with no fear of breaking down your processing performances.

### 4.5.3. Safe locks for multi-thread applications

#### 4.5.3.1. Protect your resources

Once your application is multi-threaded, concurrent data access should be protected. Otherwise, a "race condition" issue may appear: for instance, if two threads modify a variable at the same time (e.g. decrease a counter), values may become incoherent and unsafe to use. The most known symptom is the "deadlock", by which the whole application appears to be blocked and unresponsive. On a server system, which is expected to run 24/7 with no maintenance, such an issue is to be avoided.

In Delphi, protection of a resource (which may be an object, or any variable) is usually done via Critical Sections. A critical section is an object used to make sure, that some part of the code is executed only by one thread at a time. A critical section needs to be created/initialized before it can be used and be
released when it is not needed anymore. Then, some code is protected using Enter/Leave methods, which will lock its execution: in practice, only a single thread will own the critical section, so only a single thread will be able to execute this code section, and other threads will wait until the lock is released. For best performance, the protected sections should be as small as possible - otherwise the benefit of using threads may be voided, since any other thread will wait for the thread owning the critical section to release the lock.

4.5.3.2. Fixing TRTCriticalSection

In practice, you may use a TCriticalSection class, or the lower-level TRTCriticalSection record, which is perhaps to be preferred, since it will use less memory, and could easily be included as a (protected) field to any class definition.

Let's say we want to protect any access to the variables a and b. Here's how to do it with the critical sections approach:

```pascal
var CS: TRTCriticalSection;
    a, b: integer;
// set before the threads start
InitializeCriticalSection(CS);
// in each TThread.Execute:
EnterCriticalSection(CS);
try
    // protect the lock via a try ... finally block
    // from now on, you can safely make changes to the variables
    inc(a);
    inc(b);
finally
    // end of safe block
    LeaveCriticalSection(CS);
end;
// when the threads stop
DeleteCriticalSection(CS);
```

In newest versions of Delphi, you may use a TMonitor class, which will let the lock be owned by any Delphi TObject. Before XE5, there was some performance issue, and even now, this Java-inspired feature may not be the best approach, since it is tied to a single object, and is not compatible with older versions of Delphi (or FPC).

Eric Grange reported some years ago - see https://www.delphitools.info/2011/11/30/fixing-tcriticalsection - that TRTCriticalSection (along with TMonitor) suffers from a severe design flaw in which entering/leaving different critical sections can end up serializing your threads, and the whole can even end up performing worse than if your threads had been serialized. This is because it's a small, dynamically allocated object, so several TRTCriticalSection memory can end up in the same CPU cache line, and when that happens, you'll have cache conflicts aplenty between the cores running the threads.

The fix proposed by Eric is dead simple:

```pascal
type
  TFixedCriticalSection = class(TCriticalSection)
    private
      FDummy: array [0..95] of Byte;
  end;
```

4.5.3.3. Introducing TSynLocker

Since we wanted to use a TRTCriticalSection record instead of a TCriticalSection class instance, we defined a TSynLocker record in SynCommons.pas:

```pascal
TSynLocker = record
```

SAD - mORMot Framework - Rev. 1.18
As you can see, the Padding[] array will ensure that the CPU cache-line issue won't affect our object.

TSynLocker use is close to TRTLCriticalSection, with some method-oriented behavior:

```pascal
var safe: TSynLocker;
   a, b: integer;
// set before the threads start
safe.Init;
// in each TThread.Execute:
safe.Lock
try // protect the lock via a try ... finally block
   // from now on, you can safely make changes to the variables
   inc(a);
   inc(b);
finally
   // end of safe block
   safe.Unlock;
end;
// when the threads stop
safe.Done;
```

If your purpose is to protect a method execution, you may use the TSynLocker.ProtectMethod function or explicit Lock/Unlock, as such:

```pascal
type TMyClass = class
protected
   fSafe: TSynLocker;
   fField: integer;
public
   constructor Create;
   destructor Destroy; override;
   procedure UseLockUnlock;
   procedure UseProtectMethod;
end;

{ TMyClass }
constructor TMyClass.Create;
begin
   fSafe.Init; // we need to initialize the lock
end;

destructor TMyClass.Destroy;
begin
   fSafe.Done; // finalize the lock
   inherited;
end;

procedure TMyClass.UseLockUnlock;
begin
   fSafe.Lock;
   try // now we can safely access any protected field from multiple threads
      inc(fField);
   finally
      fSafe.Unlock;
   end;
```

```
procedure TMyClass.UseProtectMethod;
begin
  fSafe.ProtectMethod; // calls fSafe.Lock and return IUnknown local instance
  // now we can safely access any protected field from multiple threads
  inc(fField);
  // here fSafe.UnLock will be called when IUnknown is released
end;

4.5.3.4. Inheriting from T*Locked

For your own classes definition, you may inherit from some classes providing a TSynLocker instance, as defined in SynCommons.pas:

```pascal
TSynPersistentLocked = class(TSynPersistent)
  ...
  property Safe: TSynLocker read fSafe;
end;

TInterfacedObjectLocked = class(TInterfacedObjectWithCustomCreate)
  ...
  property Safe: TSynLocker read fSafe;
end;

TObjectListLocked = class(TObjectList)
  ...
  property Safe: TSynLocker read fSafe;
end;

TRawUTF8ListHashedLocked = class(TRawUTF8ListHashed)
  ...
  property Safe: TSynLocker read fSafe;
end;
```

All those classes will initialize and finalize their owned Safe instance, in their constructor/destructor.

So, we may have written our class as such:

```pascal
type
  TMyClass = class(TSynPersistentLocked)
  protected
    fField: integer;
  public
    procedure UseLockUnlock;
    procedure UseProtectMethod;
  end;

{ TMyClass }

procedure TMyClass.UseLockUnlock;
begin
  fSafe.Lock;
  try
    // now we can safely access any protected field from multiple threads
    inc(fField);
  finally
    fSafe.UnLock;
  end;
end;

procedure TMyClass.UseProtectMethod;
begin
  fSafe.ProtectMethod; // calls fSafe.Lock and return IUnknown local instance
  // now we can safely access any protected field from multiple threads
  inc(fField);
end;
```
As you can see, the Safe: TSynLocker instance will be defined and handled at TSynPersistentLocked parent level.

4.5.3.5. Injecting TAutoLocker instances

Inheriting from a TSynPersistentLocked class (or one of its sibling) only gives you access to a single TSynLocker per instance. If your class inherits from TSynAutoCreateFields, you may create one or several TAutoLocker published properties, which will be auto-created with the instance:

```delphi
type
  TMyClass = class(TSynAutoCreateFields)
    protected
      fLock: TAutoLocker;
      fField: integer;
    public
      function FieldValue: integer;
      published
        property Lock: TAutoLocker read fLock;
    end;
  { TMyClass }

function TMyClass.FieldValue: integer;
begin
  fLock.ProtectMethod;
  result := fField;
  inc(fField);
end;

var
  c: TMyClass;
begin
  c := TMyClass.Create;
  Assert(c.FieldValue=0);
  Assert(c.FieldValue=1);
  c.Free;
end.
```

In practice, TSynAutoCreateFields is a very powerful way of defining Value objects, i.e. objects containing nested objects or even arrays of objects. You may use its ability to create the needed TAutoLocker instances in an automated way. But be aware that if you serialize such an instance into JSON, its nested TAutoLocker properties will be serialized as void properties - which may not be the expected result.

4.5.3.6. Injecting IAutoLocker instances

If your class inherits from TInjectableObject, you may define the following:

```delphi
type
  TMyClass = class(TInjectableObject)
    private
      fLock: IAutoLocker;
      fField: integer;
    public
      function FieldValue: integer;
      published
        property Lock: IAutoLocker read fLock write fLock;
    end;
  { TMyClass }
```
function TMyClass.FieldValue: integer;
begin
    Lock.ProtectMethod;
    result := fField;
    inc(fField);
end;

var c: TMyClass;
begin
    c := TMyClass.CreateInjected([],[],[]);
    Assert(c.FieldValue=0);
    Assert(c.FieldValue=1);
    c.Free;
end;

Here we use dependency resolution - see below (page 417) - to let the TMyClass.CreateInjected constructor scan its published properties, and therefore search for a provider of IAutoLocker. Since IAutoLocker is globally registered to be resolved with TAutoLocker, our class will initialize its fLock field with a new instance. Now we could use Lock.ProtectMethod to use the associated TAutoLocker's TSynLocker critical section, as usual.

Of course, this may sounds more complicated than manual TSynLocker handling, but if you are writing an interface-based service - see below (page 419), your class may already inherit from TInjectableObject for its own dependency resolution, so this trick may be very convenient.

4.5.3.7. Safe locked storage in TSynLocker

When we fixed the potential CPU cache-line issue, do you remember that we added a padding binary buffer to the TSynLocker definition? Since we do not want to waste resource, TSynLocker gives easy access to its internal data, and allow to directly handle those values. Since it is stored as 7 slots of variant values, you could store any kind of data, including complex TDocVariant document or array.

Our class may use this feature, and store its integer field value in the internal slot 0:

type
    TMyClass = class(TSynPersistentLocked)
        public
            procedure UseInternalIncrement;
            function FieldValue: integer;
        end;
    { TMyClass }

function TMyClass.FieldValue: integer;
begin // value read will also be protected by the mutex
    result := fSafe.LockedInt64[0];
end;

procedure TMyClass.UseInternalIncrement;
begin // this dedicated method will ensure an atomic increase
    fSafe.LockedInt64Increment(0,1);
end;

Please note that we used the TSynLocker.LockedInt64Increment() method, since the following will not be safe:

procedure TMyClass.UseInternalIncrement;
begin
    fSafe.LockedInt64[0] := fSafe.LockedInt64[0]+1;
end;
In the above line, two locks are acquired (one per LockedInt64 property call), so another thread may modify the value in-between, and the increment may not be as accurate as expected.

TSynLocker offers some dedicated properties and methods to handle this safe storage. Those expect an Index value, from 0..6 range:

```pascal
property Locked[Index: integer]: Variant read GetVariant write SetVariant;
property LockedInt64[Index: integer]: Int64 read GetInt64 write SetInt64;
property LockedPointer[Index: integer]: Pointer read GetPointer write SetPointer;
property LockedUTF8[Index: integer]: RawUTF8 read GetUTF write SetUTF8;
function LockedInt64Increment(Index: integer; const Increment: Int64): Int64;
function LockedExchange(Index: integer; const Value: variant): variant;
function LockedPointerExchange(Index: integer; Value: pointer): pointer;
```

You may store a pointer or a reference to a TObject instance, if necessary.

Having such a tool-set of thread-safe methods does make sense, in the context of our framework, which offers multi-thread server abilities - see below (page 335).

### 4.5.3.8. Thread-safe TSynDictionary

Remember that the `TSynDictionary` (page 111) class is thread-safe. In fact, the TSynDictionary methods are protected by a TSynLocker instance, and internal Count or TimeOuts values are actually stored within its 7 locked storage slots.

You may consider defining TSynDictionary instances in your business logic, or in the public API layer of your services, with proper thread safety - see below (page 362).
5. Object-Relational Mapping

The ORM part of the framework - see Object-Relational Mapping (ORM) (page 92) - is mainly implemented in the mORMot.pas unit. Then it will use other units (like mORMotSQLite3.pas, mORMotDB.pas, SynSQLite3.pas or SynDB.pas) to access to the various database back-ends.

Generic access to the data is implemented by defining high-level objects as Delphi classes, descendant from a main TSQLRecord class.

In our Client-Server ORM, those TSQLRecord classes can be used for at least three main purposes:

- To store and retrieve data from any database engine - for most common usage, you can forget about writing SQL queries: CRUD data access statements (SELECT / INSERT / UPDATE /DELETE) are all created on the fly by the Object-relational mapping (ORM) core of mORMot - see below (page 239) - a NoSQL engine like MongoDB can even be accessed the same way - see below (page 287);
- To have business logic objects accessible for both the Client and Server side, in a RESTful approach - see below (page 341);
- To fill a grid content with the proper field type (e.g. grid column names are retrieved from property names after translation, enumerations are displayed as plain text, or boolean as a checkbox); to create menus and reports directly from the field definition; to have edition window generated in an automated way - see below (page 2513).

Our ORM engine has genuine advanced features like convention over configuration, integrated security, local or remote access, REST JSON publishing (for AJAX or mobile clients), direct access to the database (by-passing slow DB.pas unit), content in-memory cache, optional audit-trail (change tracking), and integration with other parts of the framework (like SOA, logging, authentication...).
5.1. TSQLRecord fields definition

All the framework ORM process relies on the TSQLRecord class. This abstract TSQLRecord class features a lot of built-in methods, convenient to do most of the ORM process in a generic way, at record level.

It first defines a primary key field, defined as ID: TID, i.e. as Int64 in mORMot.pas:

```pascal
type
TID  =  type  Int64;
...
TSQLRecord  =  class(TObject)
...
  property  ID:  TID  read  GetID  write  fID;
...
```

In fact, our ORM relies on an Int64 primary key, matching the SQLite3 ID/RowID primary key.

You may be disappointed by this limitation, which is needed by the SQLite3's implementation of Virtual Tables - see below (page 225). We won't debate about a composite primary key (i.e. several fields), which is not a good idea for an ORM. In your previous RDBMS data modeling, you may be used to define a TEXT primary key, or even a GUID primary key: those kinds of keys are somewhat less efficient than an INTEGER, especially for ORM internals, since they are not monotonic. You can always define a secondary key, as string or TGUID field, if needed - using stored AS_UNIQUE attribute as explained below.

All published properties of the TSQLRecord descendant classes are then accessed via RTTI in a Client-Server RESTful architecture.

For example, a database Baby Table is defined in Delphi code as:

```delphi
/// some enumeration
// - will be written as 'Female' or 'Male' in our UI Grid
// - will be stored as its ordinal value, i.e. 0 for sFemale, 1 for sMale
// - as you can see, ladies come first, here
TSex = (sFemale, sMale);

/// table used for the Babies queries
TSQLBaby = class(TSQLRecord)
private
  fName: RawUTF8;
  fAddress: RawUTF8;
  fBirthDate: TDateTime;
  fSex: TSex;
published
  property Name: RawUTF8  read  fName  write  fName;
  property Address: RawUTF8  read  fAddress  write  fAddress;
  property BirthDate: TDateTime  read  fBirthDate  write  fBirthDate;
  property Sex: TSex  read  fSex  write  fSex;
end;
```

By adding this TSQLBaby class to a TSQLModel instance, common for both Client and Server, the corresponding Baby table is created by the Framework in the database engine (SQLite3 natively or any external database). All SQL work ('CREATE TABLE ...') is done by the framework. Just code in Pascal, and all is done for you. Even the needed indexes will be created by the ORM. And you won't miss any ' or ; in your SQL query any more.
The following published properties types are handled by the ORM, and will be converted as specified to database content (in SQLite3, an INTEGER is an Int64, FLOAT is a double, TEXT is an UTF-8 encoded text):

<table>
<thead>
<tr>
<th>Delphi</th>
<th>SQLite3</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>word</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>cardinal</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>Int64</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>INTEGER</td>
<td>0 is false, anything else is true</td>
</tr>
<tr>
<td>enumeration</td>
<td>INTEGER</td>
<td>store the ordinal value of the enumerated item (i.e. starting at 0 for the first element)</td>
</tr>
<tr>
<td>set</td>
<td>INTEGER</td>
<td>each bit corresponding to an enumerated item (therefore a set of up to 64 elements can be stored in such a field)</td>
</tr>
<tr>
<td>single</td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td>extended</td>
<td>FLOAT</td>
<td>stored as double (precision lost)</td>
</tr>
<tr>
<td>currency</td>
<td>FLOAT</td>
<td>safely converted to/from currency type with fixed decimals, without rounding error</td>
</tr>
<tr>
<td>RawUTF8</td>
<td>TEXT</td>
<td>this is the preferred field type for storing some textual content in the ORM</td>
</tr>
<tr>
<td>WinAnsiString</td>
<td>TEXT</td>
<td>WinAnsi char-set (code page 1252) in Delphi</td>
</tr>
<tr>
<td>RawUnicode</td>
<td>TEXT</td>
<td>UCS2 char-set in Delphi, as AnsiString</td>
</tr>
<tr>
<td>WideString</td>
<td>TEXT</td>
<td>UCS2 char-set, as COM BSTR type (Unicode in all version of Delphi)</td>
</tr>
<tr>
<td>SynUnicode</td>
<td>TEXT</td>
<td>Will be either WideString before Delphi 2009, or UnicodeString later</td>
</tr>
<tr>
<td>string</td>
<td>TEXT</td>
<td>Not to be used before Delphi 2009 (unless you may lose some data during conversion) - RawUTF8 is preferred in all cases</td>
</tr>
<tr>
<td>TDateTime</td>
<td>TEXT</td>
<td>ISO 8601 encoded date time, with second resolution</td>
</tr>
<tr>
<td>TDateTimeMS</td>
<td>TEXT</td>
<td>ISO 8601 encoded date time, with millisecond resolution</td>
</tr>
<tr>
<td>TTimeLog</td>
<td>INTEGER</td>
<td>as proprietary fast Int64 date time</td>
</tr>
<tr>
<td>TModTime</td>
<td>INTEGER</td>
<td>the server date time will be stored when a record is modified (as proprietary fast Int64)</td>
</tr>
<tr>
<td>Field Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TCreateTime</td>
<td>INTEGER</td>
<td>the server date time will be stored when a record is created (as proprietary fast Int64)</td>
</tr>
<tr>
<td>TUunixTime</td>
<td>INTEGER</td>
<td>timestamp stored as second-based Unix Time (i.e. the 64-bit number of seconds since 1970-01-01 00:00:00 UTC)</td>
</tr>
<tr>
<td>TUunixMSTime</td>
<td>INTEGER</td>
<td>timestamp stored as millisecond-based Unix Time (i.e. the 64-bit number of milliseconds since 1970-01-01 00:00:00 UTC)</td>
</tr>
<tr>
<td>TSQlRecord</td>
<td>INTEGER</td>
<td>32-bit RowID pointing to another record (warning: the field value contains pointer(RowID), not a valid object instance - the record content must be retrieved with late-binding via its ID using a PtrInt(Field) typecast or the Field.ID method), or by using e.g. CreateJoined() - 64-bit under Win64</td>
</tr>
<tr>
<td>TID</td>
<td>INTEGER</td>
<td>64-bit RowID pointing to another record, but without any information about the corresponding table</td>
</tr>
<tr>
<td>TSQlRecordMany</td>
<td>nothing</td>
<td>data is stored in a separate pivot table; this is a particular case of TSQlRecord: it won't contain pointer(RowID), but an instance</td>
</tr>
<tr>
<td>TRecordReference</td>
<td>INTEGER</td>
<td>able to join any row on any table of the model, by storing both ID and TSQlRecord class type in a RecordRef-like Int64 value, with automatic reset to 0 (for TRecordReference) or row deletion (for TRecordReferenceToBeDeleted) when the pointed record is deleted</td>
</tr>
<tr>
<td>TRecordReferenceToBeDeleted</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>TSesionUserID</td>
<td>INTEGER</td>
<td>64-bit RowID of the TSQlAuthUser currently logged with the active session</td>
</tr>
<tr>
<td>TPersistent</td>
<td>TEXT</td>
<td>JSON object (ObjectToJSON)</td>
</tr>
<tr>
<td>TCollection</td>
<td>TEXT</td>
<td>JSON array of objects (ObjectToJSON)</td>
</tr>
<tr>
<td>TObjectList</td>
<td>TEXT</td>
<td>JSON array of objects (ObjectToJSON) - see TJSONSerializer. RegisterClassForJSON below (page 309)</td>
</tr>
<tr>
<td>TStrings</td>
<td>TEXT</td>
<td>JSON array of string (ObjectToJSON)</td>
</tr>
<tr>
<td>TRawUTF8List</td>
<td>TEXT</td>
<td>JSON array of string (ObjectToJSON)</td>
</tr>
<tr>
<td>any TObject</td>
<td>TEXT</td>
<td>See TJSONSerializer. RegisterCustomSerializer below (page 306)</td>
</tr>
<tr>
<td>TSQlRawBlob</td>
<td>BLOB</td>
<td>This type is an alias to RawByteString</td>
</tr>
<tr>
<td>dynamic arrays</td>
<td>BLOB</td>
<td>in the TDynArray.SaveTo binary format</td>
</tr>
<tr>
<td>variant</td>
<td>TEXT</td>
<td>numerical or text in JSON, or TDocVariant custom variant type (page 112) for JSON objects or arrays</td>
</tr>
<tr>
<td>TNullableInteger</td>
<td>INTEGER</td>
<td>Nullable Int64 value - see below (page 142)</td>
</tr>
<tr>
<td>TNullableBoolean</td>
<td>INTEGER</td>
<td>Nullable boolean (0/1/NULL) value - see below (page 142)</td>
</tr>
<tr>
<td>TNullableFloat</td>
<td>FLOAT</td>
<td>Nullable double value - see below (page 142)</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>TNullableCurrency</td>
<td>FLOAT</td>
<td>Nullable currency value - see below (page 142)</td>
</tr>
<tr>
<td>TNullableDateTime</td>
<td>TEXT</td>
<td>Nullable ISO 8601 encoded date time - see below (page 142)</td>
</tr>
<tr>
<td>TNullableTimeLog</td>
<td>INTEGER</td>
<td>Nullable TTimeLog value - see below (page 142)</td>
</tr>
<tr>
<td>TNullableUTF8Text</td>
<td>TEXT</td>
<td>Nullable Unicode text value - see below (page 142)</td>
</tr>
</tbody>
</table>

- **record**
  - TEXT
  - JSON string or object, directly handled since Delphi XE5, or as defined in code by overriding TSQLRecord. InternalRegisterCustomProperties for prior versions.

- **TRecordVersion**
  - INTEGER
  - 64-bit revision number, which will be monotonically updated each time the object is modified, to allow remote synchronization - see below (page 180)
5.1.1. Property Attributes

Some additional attributes may be added to the published field definitions:

- If the property is marked as stored AS_UNIQUE (i.e. stored false), it will be created as UNIQUE in the database (i.e. a SQL index will be created and uniqueness of the value will be checked at insert/update);
- For a dynamic array field, the index number can be used for the TSQLRecord.
  DynArray(DynArrayFieldIndex) method to create a TDynArray wrapper mapping the dynamic array data;
- For a RawUTF8 / string / WideString / WinAnsiString field of an "external" class - i.e. a TEXT field stored in a remote SynDB.pas-based database - see below (page 268), the index number will be used to define the maximum character size of this field, when creating the corresponding column in the database (SQLite3 or PostgreSQL does not have any such size expectations).

For instance, the following class definition will create an index for its SerialNumber property (up to 30 characters long if stored in an external database), and will expect a link to a model of diaper (TSQLDiaperModel) and the baby which used it (TSQLBaby). An ID / RowID column will be always available (from TSQLRecord), so in this case, you will be able to make a fast lookup for a particular diaper from either its internal mORMot ID, or its official unique serial number:

```delphi
/// table used for the Diaper queries
TSQLDiaper = class(TSQLRecord)
private
  fSerialNumber: RawUTF8;
  fModel: TSQLDiaperModel;
  fBaby: TSQLBaby;
published
  property SerialNumber: RawUTF8
    index 30
      read fSerialNumber write fSerialNumber
    stored AS.Unique;
  property Model: TSQLDiaperModel read fModel write FModel;
  property Baby: TSQLBaby read fBaby write fBaby;
end;
```

Note that TTNullableUTF8Text kind of property will follow the same index ### attribute interpretation.

5.1.2. Text fields

In practice, the generic string type is handled (as UnicodeString under Delphi 2009 and later), but you may lose some content if you're working with pre-Unicode version of Delphi (in which string = AnsiString with the current system code page). So we won't recommend its usage.

The natural Delphi type to be used for TEXT storage in our framework is RawUTF8 as introduced for Unicode and UTF-8 (page 105). All business process should better use RawUTF8 variables and methods (you have all necessary functions in SynCommons.pas), then you should explicitly convert the RawUTF8 content into a string using U2S / S2U from mORMot18n.pas or StringToUTF8 / UTF8ToString which will handle proper char-set conversion according to the current i18n settings. On Unicode version of Delphi (starting with Delphi 2009), you can directly assign a string / UnicodeString value to/from a RawUTF8, but this implicit conversion will be slightly slower than our StringToUTF8 / UTF8ToString functions. With pre-Unicode version of Delphi (up to Delphi 2007), such direct assignation will probably loose data for all non ASCII 7 bit characters, so an explicit call to StringToUTF8 / UTF8ToString functions is required.
You will find in SynCommons.pas unit all low-level RawUTF8 processing functions and classes, to be used instead of any SysUtils.pas functions. The mORMot core implementation about RawUTF8 is very optimized for speed and multi-threading, so it is recommended not to use string in your code, unless you access to the VCL / User Interface layer.

Having such a dedicated RawUTF8 type will also ensure that you are not leaking your domain from its business layer to the presentation layer, as defined with Multi-tier architecture (page 88):

For additional information about UTF-8 handling in the framework, see Unicode and UTF-8 (page 105).

5.1.3. Date and time fields

Delphi TDateTime and TDateTimeMS properties will be stored as ISO 8601 text in the database, with seconds and milliseconds resolution. See Iso8601 time and date (page 120) for details about this text encoding.

As alternatives, TTimeLog / TModTime / TCreateTime offer a proprietary fast Int64 date time format, which will map the TTimeLogBits record type, as defined in SynCommons.pas unit.

This format will be very fast for comparing dates or convert into/from text, and will be stored as INTEGER in the database, therefore more efficiently than plain ISO 8601 text as for TDateTime fields.

In practice, TModTime and TCreateTime values are inter-exchangeable with TTimeLog. They are just handled with a special care by the ORM, so that their associated field value will be updated with the current UTC timestamp, for every TSQLRecord modification (for TModTime), or at entry creation (for TCreateTime). The time value stored is in fact the UTC timestamp, as returned from the current REST Server: in fact, when any REST client perform a connection, it will retrieve any time offset from the REST Server, which will be used to store a consistent time value across all Clients.

You may also define a TUnixTime property, which will store the number of seconds since 1970-01-01 00:00:00 UTC as INTEGER in the database, and serialized as 64-bit JSON number - or TUnixMSTime if you expect milliseconds resolution. This encoding has the benefit of being handled by SQLite3 date/time functions, and interoperable with most third-party languages.

5.1.4. TSessionUserID field

If you define a TSessionUserID published property, this field will be automatically filled at creation or modification of the TSQLRecord with the current TSQLAuthUser.ID value of the active session. If no
session has been initialized from the client side, 0 will be stored.

By design, and similar to TModTime fields, you should use the ORM PUT/POST CRUD methods to compute this field value: manual SQL statements (like UPDATE Table SET Column=0) won't set its content. Also, it is up to the client to fill the TSessionUserID fields before sending their content to the server - the Delphi and cross-platform ORM clients will perform this assignment.

### 5.1.5. Enumeration fields

*Enumerations* should be mapped as INTEGER, i.e. via `ord(aEnumValue)` or `TEnum(aIntegerValue)`.

*Enumeration sets* should be mapped as INTEGER, with byte/word/integer type, according to the number of elements in the set: for instance, `byte(aSetValue)` for up to 8 elements, `word(aSetValue)` for up to 16 elements, and `integer(aSetValue)` for up to 32 elements.

### 5.1.6. Floating point and Currency fields

For standard floating-point values, the framework natively handles the double and currency kind of variables.

In fact, `double` is the native type handled by most database providers - it is also native to the SSE set of opcodes of newer CPUs (as handled by Delphi XE 2 in 64-bit mode). Lack of extended should not be problematic (if it is mandatory, a dedicated set of mathematical classes should be preferred to a database), and could be implemented with the expected precision via a TEXT field (or a BLOB mapped by a dynamic array).

The currency type is the standard *Delphi* type to be used when storing and handling monetary values, native to the x87 FPU - when it comes to money, a dedicated type is worth the cost in a "rich man's world". It will avoid any rounding problems, assuming exact 4 decimals precision. It is able to safely store numbers in the range `−922337203685477.5808 .. 922337203685477.5807`. Should be enough for your pocket change.

As stated by the official *Delphi* documentation:

*Currency* is a fixed-point data type that minimizes rounding errors in monetary calculations. On the Win32 platform, it is stored as a scaled 64-bit integer with the four least significant digits implicitly representing decimal places. When mixed with other real types in assignments and expressions, Currency values are automatically divided or multiplied by 10000.

In fact, this type matches the corresponding OLE and .Net implementation of currency. It is still implemented the same in the *Win64* platform (since XE 2). The `Int64` binary representation of the currency type (i.e. `value*10000` as accessible via a typecast like `PInt64(@aCurrencyValue)^`) is a safe and fast implementation pattern.

In our framework, we tried to avoid any unnecessary conversion to float values when dealing with currency values. Some dedicated functions have been implemented - see *Currency handling* (page 106) - for fast and secure access to currency published properties via RTTI, especially when converting values to or from JSON text. Using the `Int64` binary representation can be not only faster, but also safer: you will avoid any rounding problem which may be introduced by the conversion to a float type. For all database process, especially with external engines, the SynDB.pas units will try to avoid any conversion to/from `double` for the dedicated `ftCurrency` columns.

Rounding issues are a nightmare to track in production - it sounds safe to have a framework handling natively a currency type from the ground up.
5.1.7. TSQLRecord fields

It is worth saying that TSQLRecord published properties are not by default class instances, as with regular Delphi code. After running TSQLRecord.Create() or CreateAndFillPrepare() constructors, you should never call aMyRecord.AnotherRecord.Property directly, or you will raise an Access Violation.

In fact, TSQLRecord published properties definition is used to define "one to many" or "one to one" relationships between tables. As a consequence, the nested AnotherRecord property won’t be a true class instance, but one ID trans-typed as TSQLRecord.

Only exception to this rule is TSQLRecordMany kind of published properties, which, by design, are true instances, needed to access the pivot table data of "many to many" relationship. The ORM will auto-instantiate all TSQLRecordMany published properties, then release them at Destroy - so you do not need to maintain their life time.

Note that you may use e.g. TSQLRecord.CreateJoined() constructor to auto-instantiate and load all TSQLRecord published properties at once, then release them at Destroy. - see below (page 153).

The ORM will automatically perform the following optimizations for TSQLRecord published fields:
- An index will be created on the database, for the corresponding column;
- When a referenced record is deleted, the ORM will detect it and automatically set all published properties pointing to this record to 0.

In fact, the ORM won’t define a ON DELETE SET DEFAULT foreign key via SQL: this feature won’t be implemented at RDBMS level, but emulated at ORM level.

See below (page 151) for more details about how to work with TSQLRecord published properties.

5.1.8. TID fields

TSQLRecord published properties do match a class instance pointer, so are 32-bit (at least for Win32/Linux32 executables). Since the TSQLRecord.ID field is declared as TID = Int64, we may lose information if the stored ID is greater than 2,147,483,647 (i.e. a signed 32-bit value).

You can define a published property as TID to store any value of our primary key, i.e. up to 9,223,372,036,854,775,808. Note that in this case, there is no information about the joined table.

As a consequence, the ORM will perform the following optimizations for TID fields:
- An index will be created on the database, for the corresponding column;
- When a referenced record is deleted, the ORM won’t do anything, since it has no information about the table to track - this is the main difference with TSQLRecord published property.

You can optionally specify the associated table, using a custom TID type for the published property definition. In this case, you will sub-class TID, using tableNameID as naming convention.

For instance, if you define:

```
  type
  TSQLRecordClientID = type TID;
  TSQLRecordClientToBeDeletedID = type TID;

  TSQLOrder = class(TSQLRecord)
    ...
    property Client: TID
      read fClient write fClient;
    property OrderedBy: TSQLRecordClientID
      read fOrderedBy write fOrderedBy;
```
property OrderedByCascade: TSQLRecordClientToBeDeletedID
        read fOrderedByCascade write fOrderedByCascade;

Those three published fields will be able to store a Int64 foreign key, and the ORM will ensure a corresponding index is created on the database, to speedup search on their values.
But their type - TID, TSQLRecordClientID, or TSQLRecordClientToBeDeletedID - will define how the deletion process will be processed.

By using the generic TID type, the first Client property won't have any reference to any table, so no deletion tracking will take place.

On the other hand, following the type naming convention, the others OrderedBy and OrderedByCascade properties will be associated with the TSQLRecordClient table of the data model.
In fact, the ORM will retrieve the 'TSQLRecordClientID' or 'TSQLRecordClientToBeDeletedID' type names, and search for a TSQLRecord associated by trimming *[ToBeDeleted]ID, which is TSQLRecordClient in this case.
As a result, the ORM will be able to track any TSQLRecordClient deletion: for any row pointing to the deleted record, it will ensure that this OrderedBy property will be reset to 0, or that the row containing the OrderedByCascade property will be deleted. Note that the framework won't define a ON DELETE SET DEFAULT or ON DELETE CASCADE foreign key via SQL, but emulate them at ORM level.

5.1.9. TRecordReference and TRecordReferenceToBeDeleted

TSQLRecord or TID published properties are associated with a single TSQLRecord joined table. You could use TRecordReference or TRecordReferenceToBeDeleted published properties to store a reference to any record on any table of the data model.

In fact, such properties will store in a Int64 value a reference to both a TSQLRecord class (therefore defining a table), and one ID (to define the row).

You could later on use e.g. TSQLRest.Retrieve(Reference) to get a record content in one step.

One important note is to remember that the table reference is stored as an index to the TSQLRecord class in the associated TSQLModel.
As a consequence, for such TRecordReference* properties to work as expected, you should ensure:
- That the order of TSQLRecord classes in the TSQLModel do not change after any model modification: otherwise, all previously stored TRecordReference* values may point to a wrong record;
- That both Client and Server side share the same model - at least for the TSQLRecord classes which are used with TRecordReference*.

Depending on the type, the ORM will track the deletion of the pointed record:
- TRecordReference fields will be reset to 0 - emulating ON DELETE SET DEFAULT foreign key SQL declaration;
- TRecordReferenceToBeDeleted will delete the whole record - emulating ON DELETE CASCADE foreign key SQL declaration.

Just like with TSQLRecord or TSQLRecordClassName[ToBeDeleted]ID fields, this deletion tracking is not defined at RDBMS level, but emulated at ORM level.

In order to work easily with TRecordReference values (which are in fact plain Int64 values), you could transtype them into the RecordRef() record, and access the stored information via a set of helper methods. See below (page 2507) for an example of use of such TRecordReference in a data model, e.g. the AssociatedRecord property of TSQLAuditTrail.
5.1.10. TSQLRecord, TID, TRecordReference deletion tracking

To sum up all possible foreign key reference available by the framework, check out this table:

<table>
<thead>
<tr>
<th>Type Definition</th>
<th>Index</th>
<th>Tables</th>
<th>Deletion Tracking</th>
<th>Emulated SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLRecord</td>
<td>Yes</td>
<td>One</td>
<td>Field reset to 0</td>
<td>ON DELETE SET DEFAULT</td>
</tr>
<tr>
<td>TID</td>
<td>Yes</td>
<td>No</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>TClassName</td>
<td>Yes</td>
<td>One</td>
<td>Field reset to 0</td>
<td>ON DELETE SET DEFAULT</td>
</tr>
<tr>
<td>TClassNameToBeDeletedID</td>
<td>Yes</td>
<td>One</td>
<td>Row deleted</td>
<td>ON DELETE CASCADE</td>
</tr>
<tr>
<td>TRecordReference</td>
<td>Yes</td>
<td>All</td>
<td>Field reset to 0</td>
<td>ON DELETE SET DEFAULT</td>
</tr>
<tr>
<td>TRecordReferenceToBeDeleted</td>
<td>Yes</td>
<td>All</td>
<td>Row deleted</td>
<td>ON DELETE CASCADE</td>
</tr>
</tbody>
</table>

It is worth saying that this deletion tracking is not defined at RDBMS level, but at ORM level. As a consequence, it will work with any kind of databases, including NoSQL and Object-Document Mapping (ODM) (page 96). In fact, RDBMS engines do not allow defining such ON DELETE trigger on several tables, whereas mORMot handles such composite references as expected for TRecordReference. Since this is not a database level tracking, but only from a mORMot server, if you still use the database directly from legacy code, ensure that you will take care of this tracking, perhaps by using a SOA service instead of direct SQL statements.

5.1.11. Variant fields

The ORM will store variant fields as TEXT in the database, serialized as JSON.

At loading, it will check their content:
- If some custom variant types are registered (e.g. MongoDB custom objects), they will be recognized as such (with extended syntax, if applying);
- It will create a TDocVariant custom variant type (page 112) instance if the stored TEXT is a JSON object or array;
- It will create a numerical value (integer or double) if the stored text has the corresponding layout;
- Otherwise, it will create a string value.

Since all data is stored as TEXT in the column, your queries shall ensure that any SQL WHERE statement handles it as expected (e.g. with a conversion to number before comparison). Even if SQLite3 is able to affect a column type for each row (i.e. store a variant as in Delphi code), we did not use this feature, since we wanted our framework to work with all databases - and SQLite3 is quite alone having this feature.

At JSON level, variant fields will be transmitted as JSON text or number, depending on the stored value.

If you use a MongoDB external NoSQL database - see below (page 287), such variant field will not be stored as JSON text, but as true BSON documents. So you will be able to apply all the advanced search and indexing abilities of this database engine, if needed.

5.1.12. Record fields

Since Delphi XE5, you can define and work directly with published record properties of TSQLRecord:
The record will be serialized as JSON - here TGUID will be serialized as a JSON string - then will be stored as TEXT column in the database.

We specified an index 38 attribute to state that this column will contain up to 38 characters, when stored on an external database - see below (page 268).

Published properties of records are handled by our code, but Delphi doesn't create the corresponding RTTI for such properties before Delphi XE5. So record published properties, as defined in the above class definition, won't work directly for older versions of Delphi, or FreePascal.

You could use a dynamic array with only one element, in order to handle records within your TSQLRecord class definition - see below (page 161). But it may be confusing.

If you want to work with such properties before Delphi XE5, you can override the TSQLRecord.InternalRegisterCustomProperties() virtual method of a given table, to explicitly define a record property.

For instance, to register a GUID property mapping a TSQLMyRecord.fGUID: TGUID field:

```pascal
TSQLMyRecord = class(TSQLRecordPeople)
protected
  fGUID: TGUID;
published
  property GUID: TGUID read fGUID write fGUID index 38;
end;

{ TSQLMyRecord }

class procedure TSQLMyRecord.InternalRegisterCustomProperties(
  Props: TSQLRecordProperties);
begin
  Props.RegisterCustomPropertyFromTypeName(self,'TGUID','GUID',
    @TSQLRecordCustomProps(nil).fGUID,[aIsUnique],38);
end;
```

You may call Props.RegisterCustomPropertyFromRTTI(), supplying the TypeInfo() pointer, for a record containing reference-counted fields like string, variant or nested dynamic arrays. Of course, any custom JSON serialization of the given record type - see below (page 297) - will be supported.

Those custom record registration methods will define either:
- TEXT serialization, for RegisterCustomPropertyFromRTTI() or RegisterCustomPropertyFromTypeName();
- BLOB serialization, for RegisterCustomRTTIRecordProperty() or RegisterCustomFixedSizeRecordProperty().

### 5.1.13. BLOB fields

In fact, several kind of properties will be stored as BLOB in the database backend:
- TSQLRawBlob properties are how you store your binary data, e.g. images or documents;
- dynamic arrays (saved in the TDynArray.SaveTo binary format);
- record which were explicitly registered as BLOB columns.

By default, both *dynamic arrays* and BLOB *record* content will be retrieved from the database, encoded as Base64 text.

But TSQLRawBlob properties will be transmitted as RESTful separate resources, as required by the REST scheme. For instance, it means that a first request will retrieve all "simple" fields as JSON, then some other requests are needed to retrieve each BLOB fields as a binary buffer. As a result, TSQLRawBlob won't be transmitted by default, to spare transmission bandwidth and resources.

You can change this default behavior, by setting:
- Either TSQLRestClientURI.ForceBlobTransfer: boolean property, to force the transfer of all BLOBs of all the tables of the data model - this is what is done e.g. for the SynFile main demo - see later in this document;
- Or via TSQLRestClientURI.TSQLRestClientURI.ForceBlobTransferTable[] property, for a specified table of the model.

### 5.1.14. TNullable* fields for NULL storage

In Delphi, nullable types do not exist, as they do for instance in C#, via the `int?` kind of definition. But at SQL and JSON levels, the NULL values do exist and are expected to be available from our ORM.

In SQLite3 itself, NULL is handled as stated in [http://www.sqlite.org/lang_expr.html](http://www.sqlite.org/lang_expr.html) (see e.g. IS and IS NOT operators).

It is worth noting that NULL handling is not consistent among all existing database engines, e.g. when you are comparing NULL with non NULL values... so we recommend using it with care in any database statements, or only with proper (unit) testing, when you switch from one database engine to another.

By default, in the mORMot ORM/SQL code, NULL will appear only in case of a BLOB storage with a size of 0 bytes. Otherwise, you should not see it as a value, in most used types - see *TSQLRecord fields definition* (page 131).

Null-oriented value types have been implemented in our framework, since the object pascal language does not allow defining a nullable type (yet). We choose to store those values as variant, with a set of TNullable* dedicated types, as defined in mORMot.pas:

```pascal
{type}
TNullableInteger = type variant;
TNullableBoolean = type variant;
TNullableFloat = type variant;
TNullableCurrency = type variant;
TNullableDateTime = type variant;
TNullableTimeLog = type variant;
TNullableUTF8Text = type variant;
{type}
TSQLNullableRecord = class(TSQLRecord)
protected
  fInt: TNullableInteger;
fBool: TNullableBoolean;
ffLt: TNullableFloat;
fCurr: TNullableCurrency;
fDate: TNullableDateTime;
ftimeStamp: TNullableTimeLog;
fCLOB: TNullableUTF8Text;
fText: TNullableUTF8Text;
```
published
property Int: TNullableInteger read fInt write fInt;
property Bool: TNullableBoolean read fBool write fBool;
property Flt: TNullableFloat read fFlt write fFlt;
property Curr: TNullableCurrency read fCurr write fCurr;
property Date: TNullableDateTime read fDate write fDate;
property Timestamp: TNullableTimeLog read fTimestamp write fTimestamp;
property CLOB: TNullableUTF8Text read fCLOB write fCLOB;
property Text: TNullableUTF8Text index 32 read fText write fText;
end;

Such a class will let the ORM handle SQL NULL values as expected, i.e. returning a null variant value, or an integer/number/text value if there is something stored. Of course, the corresponding column in the database will have the expected data type, e.g. a NULLABLE INTEGER for TNullableInteger property.

Note that TNullableUTF8Text is defined as a RawUTF8 usual field - see Property Attributes (page 135). That is, without any size limitation by default (as for the CLOB property), or with an explicit size limitation using the index ### attribute (as for Text property, which will be converted as a VARCHAR(32) SQL column).

You could use the following wrapper functions to create a TNullable* value from any non-nullable standard Delphi value:

function NullableInteger(const Value: Int64): TNullableInteger;
function NullableBoolean(Value: boolean): TNullableBoolean;
function NullableFloat(const Value: double): TNullableFloat;
function NullableCurrency(const Value: currency): TNullableCurrency;
function NullableDateTime(const Value: TDateTime): TNullableDateTime;
function NullableTimeLog(const Value: TTimeLog): TNullableTimeLog;
function NullableUTF8Text(const Value: RawUTF8): TNullableUTF8Text;

Some corresponding constants do match the expected null value for each kind, with strong typing (to be used for FPC compatibility, which does not allow direct assignment of a plain null: variant to a TNullable* = type variant property):

var
NullableIntegerNull: TNullableInteger absolute NullVarData;
NullableBooleanNull: TNullableBoolean absolute NullVarData;
...

You could check for a TNullable* value to contain null, using the following functions:

function NullableIntegerIsEmptyOrNull(const V: TNullableInteger): Boolean;
function NullableBooleanIsEmptyOrNull(const V: TNullableBoolean): Boolean;
...

Or retrieve a Delphi non-nullable value in one step, using the corresponding wrappers:

function NullableIntegerToValue(const V: TNullableInteger; out Value: Int64): Boolean;
function NullableBooleanToValue(const V: TNullableBoolean; out Value: Boolean): Boolean;
...

Those Nullable*ToValue() functions are mandatory for use under FPC, which does not allow mixing plain variant values and specialized TNullable* = type variant values.

Thanks to those types, and their corresponding wrapper functions, you have at hand everything needed to safely store some nullable values into your application database, with proper handling on Delphi side.
5.2. Working with Objects

To access a particular record, the following code can be used to handle CRUD statements (Create Retrieve Update Delete actions are implemented via Add/Update/Delete/Retrieve methods), following the RESTful pattern - see below (page 311), and using the ID primary key as resource identifier:

```pascal
procedure Test(Client: TSQLRest); // we will use CRUD operations on a REST instance
var Baby: TSQLBaby; // store a record
    ID: TID; // store a reference to a record
begin
    // create and save a new record, since Smith, Jr was just born
    Baby := TSQLBaby.Create;
    try
        Baby.Name := 'Smith';
        Baby.Address := 'New York City';
        Baby.BirthDate := Date;
        Baby.Sex := sMale;
        ID := Client.Add(Baby,true);
    finally
        Baby.Free; // manage memory as usual
    end;
    // update record data
    Baby := TSQLBaby.Create(Client,ID); // retrieve from ID
    try
        assert(Baby.Name='Smith');
        Baby.Name := 'Smeeth';
        Client.Update(Baby);
    finally
        Baby.Free;
    end;
    // retrieve record data
    Baby := TSQLBaby.Create;
    try
        Client.Retrieve(ID,Baby);
        // we may have written:  Baby := TSQLBaby.Create(Client,ID);
        assert(Baby.Name='Smeeth');
    finally
        Baby.Free;
    end;
    // delete the created record
    Client.Delete(TSQLBaby,ID);
end;
```

Of course, you can have a TSQLBaby instance alive during a longer time. The same TSQLBaby instance can be used to access several record content, and call Retrieve / Add / Delete / Update methods on purpose.

No SQL statement to write, nothing to care about database engine expectations (e.g. for date or numbers processing): just accessing objects via high-level methods. It could even work with NoSQL databases, like a fast TObjectList or MongoDB. This is the magic of ORM.

To be honest, the REST pattern does not match directly the CRUD operations exactly. We had to tied a little bit the REST verbs - as defined below (page 311) - to fit our ORM purpose. But all you have to know is that those Add/Update/Delete/Retrieve methods are able to define the full persistence lifetime of your precious objects.

5.3. Queries

5.3.1. Return a list of objects

You can query your table with the FillPrepare or CreateAndFillPrepare methods, for instance all
babies with balls and a name starting with the letter 'A':

```pascal
var aMale: TSQLBaby;
...
aMale := TSQLBaby.CreateAndFillPrepare(Client,
  [Name LIKE ? AND Sex = ?],["A%",ord(sMale)]);
try
  while aMale.FillOne do
    DoSomethingWith(aMale);
finally
  aMale.Free;
end;
```

This request loops through all matching records, accessing each row content via a TSQLBaby instance.

The mORMot engine will create a SQL statement with the appropriate SELECT query, retrieve all data as JSON, transmit it between the Client and the Server (if any), then convert the values into properties of our TSQLBaby object instance. Internally, the [CreateAnd]FillPrepare / FillOne methods use a list of records, retrieved as JSON from the Server, and parsed in memory one row a time (using an internal TSQLTableJSON instance).

Note that there is an optional aCustomFieldsCSV parameter available in all FillPrepare / CreateAndFillPrepare methods, by which you may specify a CSV list of field names to be retrieved. It may save some remote bandwidth, if not all record fields values are needed in the loop. Note that you should use this aCustomFieldsCSV parameter only to retrieve some data, and that the other fields will remain untouched (i.e. void in case of CreateAndFillPrepare): any later call to Update should lead into a data loss, since the method will know that is has been called during a FillPrepare / CreateAndFillPrepare process, and only the retrieved filled will be updated on the server side.

You could also create a TObjectList, or - even better for newer versions of Delphi supporting the generics syntax - a TObjectList<T> instance to retrieve all values of a table:

```pascal
var aList: TObjectList<TSQLBaby>;
aMale: TSQLBaby;
...
```

```pascal
try
  for aMale in aList do
    DoSomethingWith(aMale);
finally
  aList.Free;
end;
```

Note that this method will use more memory and resources than a *FillPrepare call followed by a while ...FillOne do loop, since the later will only allocate one instance of the TSQLRecord, then fill the properties of this single instance directly from the returned JSON content, one at a time. For huge lists, or in multi-threaded environment, it may make a difference.

But the generics syntax can make cleaner code, or more integrated with your business logic.

### 5.3.2. Query parameters

For safer and faster database process, the WHERE clause of the request expects some parameters to be specified. They are bound in the ? appearance order in the WHERE clause of the [CreateAnd]FillPrepare query method.

Standard simple kind of parameters (RawUTF8, integer, double, currency..) can be bound directly - as in the sample code above for Name or Sex properties. The first parameter will be bound as 'A%' RawUTF8 TEXT, and the second as the 1 INTEGER value.
Any TDateTime bound parameter shall better be specified using DateToSQL(), DateTimeToSQL() or TimeLogToSQL() functions, as such:

```pascal
aRec.CreateAndFillPrepare(Client,'Datum=?',[DateToSQL(EncodeDate(2012,5,4))]);
aRec.CreateAndFillPrepare(Client,'Datum>=?',[DateToSQL(2012,5,4)]);
aRec.CreateAndFillPrepare(Client,'Datum<=?',[DateTimeToSQL(Now)]);
aRec.CreateAndFillPrepare(Client,'Datum<=?',[TimeLogToSQL(Client.ServerTimestamp)]);
```

For TTimeLog / TModTime / TCreateTime / TUunixTime / TUunixMSTime kind of properties, please use the underlying Int64 value as bound parameter.

As stated previously, BLOB (i.e. sftBlob or TSQLRawBlob) properties are handled separately, via dedicated RetrieveBlob and UpdateBlob method calls (or their global RetrieveBlobFields / UpdateBlobFields twins). In fact, BLOB data is expected to be potentially big (more than a few MB). But you can specify a small BLOB content using an explicit conversion to the corresponding TEXT format, by calling BinToBase64WithMagic() overloaded functions when preparing an UPDATE query, or by defining a TByteDynArray published field instead of TSQLRawBlob.

See also ForceBlobTransfer and ForceBlobTransferTable[] properties of TSQLRestClientURI.

Note that there was a breaking change about the TSQLRecord.Create / FillPrepare / CreateAndFillPrepare and TSQLRest.OneFieldValue / MultiFieldValue methods: for historical reasons, they expected parameters to be marked as % in the SQL WHERE clause, and inlined via :(...) as stated below (page 148) - since revision 1.17 of the framework, those methods expect parameters marked as ? and with no :(...):: Due to this breaking change, user code review is necessary if you want to upgrade the engine from 1.16 or previous. In all cases, using ? is less confusing for new users, and more close to the usual way of preparing database queries - e.g. as stated below (page 239). Both TSQLRestClient.ExecuteFmt / ListFmt methods are not affected by this change, since they are just wrappers to the FormatUTF8() function.

For the most complex codes, you may want to prepare ahead the WHERE clause of the ORM request. You may use the overloaded FormatUTF8() function as such:

```pascal
var where: RawUTF8;
begin
  where := FormatUTF8('id=?', [], [SomeID]);
  if add_active then
    where := FormatUTF8('% and active=?', [where], [ActiveFlag]);
  if add_date_ini then
    where := FormatUTF8('% and date_ini>=?',[where], [DateToSQL(Date-2)]);
...
```

Then the request will be easy to create, and fast to execute, thanks to prepared statements in the framework database layer.

### 5.3.3. Introducing TSQLTableJSON

As we stated above, [CreateAnd]FillPrepare / FillOne methods are implemented via an internal TSQLTableJSON instance.

In short, TSQLTableJSON will expect some JSON content as input, will parse it in rows and columns, associate it with one or more optional TSQLRecord class types, then will let you access the data via its Get* methods.

You can use this TSQLTableJSON class as in the following example:

```pascal
procedure WriteBabiesStartingWith(const Letters: RawUTF8; Sex: TSex);
var alist: TSQLTableJSON;
  Row: integer;
begin
```
For a record with a huge number of fields, specifying the needed fields could save some bandwidth. In the above sample code, the ID column has a field index of 0 (so is retrieved via `aList.GetAsInteger(Row,0)`) and the BirthDate column has a field index of 1 (so is retrieved as a `PUTF8Char` via `aList.Get(Row,1)`). All data rows are processed via a loop using the `RowCount` property count - first data row is indexed as 1, since the row 0 will contain the column names.

The `TSQLTable` class has some methods dedicated to direct cursor handling, as such:

```pascal
procedure WriteBabiesStartingWith(const Letters: RawUTF8; Sex: TSex);
var aList: TSQLTableJSON;
begin
  aList := Client.MultiFieldValues(TSQLBaby, 'ID,BirthDate', 'Name LIKE ? AND Sex = ?', [Letters+'%',ord(Sex)]);
  try
    while aList.Step do
      writeln('ID=',aList.Field('ID'),' BirthDate=',aList.Field('BirthDate'));
  finally
    aList.Free;
  end;
end;
```

By using the `TSQLTable`.Step method, you do not need to check that `aList<>nil`, since it will return false if `aList` is not assigned. And you do not need to access the `RowCount` property, nor specify the current row number.

We may have used not the field index, but the field name, within the loop:

```pascal
writeln('ID=',aList.Field('ID'),' BirthDate=',aList.Field('BirthDate'));
```

You can also access the field values using late-binding and a local variant, which gives some perfectly readable code:

```pascal
procedure WriteBabiesStartingWith(const Letters: RawUTF8; Sex: TSex);
var baby: variant;
begin
  with Client.MultiFieldValues(TSQLBaby,'ID,BirthDate', 'Name LIKE ? AND Sex = ?', [Letters+'%',ord(Sex)]) do
  try
    while Step(false,@baby) do
      writeln('ID=',baby.ID,' BirthDate=',baby.BirthDate);
  finally
    Free;
  end;
end;
```

In the above code, late-binding will search for the "ID" and "BirthDate" fields at runtime. But the ability to write `baby.ID` and `baby.BirthDate` is very readable. Using a `with ... do` statement makes the code shorter, but should be avoided if it leads into confusion, e.g. in case of more complex process within the loop.

See also the following methods of `TSQLRest`: `OneFieldValue`, `OneFieldValues`, `MultiFieldValue`, `MultiFieldValues` which are able to retrieve either a `TSQLTableJSON`, or a `dynamic array` of integer or `RawUTF8`. And also `List` and `ListFmt` methods of `TSQLRestClient`, if you want to make a JOIN
against multiple tables at once.

A TSQLTableJSON content can be associated to a TGrid in order to produce an User Interface taking advantage of the column types, as retrieved from the associated TSQLRecord RTTI. The TSQLTableToGrid class is able to associate any TSQLTable to a standard TDrawGrid, with some enhancements: themed drawing, handle Unicode, column types (e.g. boolean are displayed as check-boxes, dates as text, etc...), column auto size, column sort, incremental key lookup, optional hide IDs, selection...

5.3.4. Note about query parameters

*(This paragraph is not mandatory to be read at first, so you can skip it if you do not need to know about the mORMot internals - just remember that ? bound parameters are inlined as :(...) in the JSON transmitted content so can be set directly as such in any WHERE clause)*

If you consider the first sample code:

```pascal
aMale := TSQLBaby.CreateAndFillPrepare(Client,
    'Name LIKE ? AND Sex = ?', ['A%', ord(sMale)]);
```

This will execute a SQL statement, with an ORM-generated SELECT, and a WHERE clause using two parameters bound at execution, containing 'A%' RawUTF8 text and 1 integer value.

In fact, from the SQL point of view, the CreateAndFillPrepare() method as called here is exactly the same as:

```pascal
aMale := TSQLBaby.CreateAndFillPrepare(Client,
    'Name LIKE :(''A%''): AND Sex = :((1)));
```

or

```pascal
aMale := TSQLBaby.CreateAndFillPrepare(Client,
    'Name LIKE :(%): AND Sex = :(%):', ['','A%',''], [1]));
```

or

```pascal
aMale := TSQLBaby.CreateAndFillPrepare(Client,
    FormatUTF8('Name LIKE :(%): AND Sex = :(%):', ['','A%''',''], ord(sMale)));
```

First point is that the 'A' letter has been embraced with quotes, as expected per the SQL syntax. In fact, `Name LIKE :(%): AND Sex = :(%):', ['''A%''',''], ord(sMale)]` is expected to be a valid WHERE clause of a SQL statement.

Note we used single quotes, but we may have used double quotes (") inside the :(...) statements. In fact, SQLite3 expects single quotes in its raw SQL statements, whereas our prepared statements :(...) will handle both single ' and double " quotes. Just to avoid any confusion, we'll always show single quotes in the documentation. But you can safely use double quotes within :(...) statements, which could be more convenient than single quotes, which should be doubled within a pascal constant string ".

The only not-obvious syntax in the above code is the :(...) used for defining prepared parameters in the format string.

In fact, the format string will produce the following WHERE clause parameter as plain text:

```pascal
aMale := TSQLBaby.CreateAndFillPrepare(Client,
    'Name LIKE :(''A%''): AND Sex = :((1)));
```

So that the following SQL query will be executed by the database engine, after translation by the ORM magic:
```sql
SELECT * FROM Baby WHERE Name LIKE ? AND Sex = ?;
```

With the first ? parameter bound with 'A%' value, and the second with 1.

In fact, when the framework finds some :( ): in the SQL statement string, it will prepare a SQL statement, and will bind the parameters before execution (in our case, text A and integer 1), reusing any previous matching prepared SQL statement. See below (page 213) for more details about this mechanism.

To be clear, without any prepared statement, you could have used:

```pascal
aMale := TSQLBaby.CreateAndFillPrepare(Client,
   'Name LIKE % AND Sex = %','["%A%",ord(sMale)],[],);
```

or

```pascal
aMale := TSQLBaby.CreateAndFillPrepare(Client,
   FormatUTF8('Name LIKE % AND Sex = %','["%A%",ord(sMale)]));
```

which will produce the same as:

```pascal
aMale := TSQLBaby.CreateAndFillPrepare(Client,
   'Name LIKE ''A%' AND Sex = 1');
```

So that the following SQL statement will be executed:

```sql
SELECT * FROM Baby WHERE Name LIKE 'A%' AND Sex = 1;
```

Note that we prepared the SQL WHERE clause, so that we could use the same request statement for all females with name starting with the character 'D':

```pascal
aFemale := TSQLBaby.CreateAndFillPrepare(Client,
   'Name LIKE :(%): AND Sex = :(%):',[D%',ord(sFemale)]);
```

Using a prepared statement will speed up the database engine, because the SQL query will have to be parsed and optimized only once.

The second query method, i.e.

```pascal
aList := Client.MultiFieldValues(TSQLBaby,'ID,BirthDate',
   'Name LIKE ? AND Sex = ?',[Letters+%',ord(Sex)]);
```

is the same as this code:

```pascal
aList := Client.MultiFieldValues(TSQLBaby,'ID,BirthDate',
   'Name LIKE :(%): AND Sex = :(%):',[QuotedStr(Letters+%'),ord(Sex)],[],);
```

or

```pascal
aList := Client.MultiFieldValues(TSQLBaby,'ID,BirthDate',
   FormatUTF8('Name LIKE :(%): AND Sex = :(%):',[QuotedStr(Letters+%'),ord(Sex)]));
```

In both cases, the parameters will be inlined, in order to prepare the statements, and improve execution speed.

We used the QuotedStr standard function to embrace the Letters parameter with quotes, as expected per the SQL syntax.

Of course, using '?' and bounds parameters is much easier than '%' and manual :(%): in-lining with a QuotedStr() function call. In your client code, you should better use '?' - but if you find some ':'(%): in the framework source code and when a WHERE clause is expected within the transmitted JSON content, you won't be surprised.

### 5.4. Automatic TSQLRecord memory handling

Working with objects is pretty powerful, but requires to handle manually the created instances life
time, via try .. finally blocks. Most of the time, the TSQLRecord life time will be very short: we allocate one instance on a local variable, then release it when it goes out of scope.

If we take again the TSQLBaby sample, we may write:

```pascal
function NewMaleBaby(Client: TSQLRest; const Name, Address: RawUTF8): TID;
var Baby: TSQLBaby; // store a record
begin
    Baby := TSQLBaby.Create;
    try
        Baby.Name := Name;
        Baby.Address := Address;
        Baby.BirthDate := Date;
        Baby.Sex := sMale;
        result := Client.Add(Baby, true);
    finally
        Baby.Free;
    end;
end;
```

To ease this pretty usual pattern, the framework offers some kind of automatic memory management at TSQLRecord level:

```pascal
function NewMaleBaby(Client: TSQLRest; const Name, Address: RawUTF8): TID;
var Baby: TSQLBaby;
begin
    TSQLBaby.AutoFree(Baby); // no try..finally needed!
    Baby.Name := Name;
    Baby.Address := Address;
    Baby.BirthDate := Date;
    Baby.Sex := sMale;
    result := Client.Add(Baby, true);
end; // Local Baby instance will be released here
```

It may also be useful for queries.

Instead of writing:

```pascal
var aMale: TSQLBaby;
...
aMale := TSQLBaby.CreateAndFillPrepare(Client,
    Name LIKE ? AND Sex = ?,['A%'],ord(sMale));
try
    while aMale.FillOne do
        DoSomethingWith(aMale);
finally
    aMale.Free;
end;
```

We may write:

```pascal
var aMale: TSQLBaby;
...
TSQLBaby.AutoFree(aMale,Client,'Name LIKE ? AND Sex = ?',['A%'],ord(sMale));
while aMale.FillOne do
    DoSomethingWith(aMale);
```

Without the need to write the try ... finally block.

See the TSQLRecord.AutoFree() overloaded methods in mORMot.pas for the several use cases, and the associated TAutoFree / IAutoFree types as defined in SynCommons.pas. Note that you can handle several local variables in a single TSQLRecord.AutoFree() or TAutoFree.Create() initialization.

Be aware that it does not introduce some kind of magic garbage collector, as available in C# or Java. It is not even similar to the ARC memory model used by Apple and the Delphi NextGen compiler. It is just some syntactic sugar creating a local hidden IAutoFree interface, which will be released at the end of
the local method by the compiler, and also release all associated class instances. So the local class instances should stay in the local scope, and should not be sent and stored in another process: in such cases, you may encounter access violation issues.

Due to an issue (feature?) in the FPC implementation of interfaces - see http://bugs.freepascal.org/view.php?id=26602. - the above code will not work directly. You should assign the result of this method to a local IAutoFree variable, as such:

```pascal
var aMale: TSQLBaby;
    auto: IAutoFree;

    'auto := TSQLBaby.AutoComplete(aMale,Client,'Name LIKE ? AND Sex = ?',['A%',ord(sMale)]);
    while aMale.FillOne do
        DoSomethingWith(aMale);
```

One alternative may be to use a with statement, which prevents the need of defining a local variable:

```pascal
var aMale: TSQLBaby;

    with TAutoFree.One(aMale,TSQLBaby.CreateAndFillPrepare(Client,
        'Name LIKE ? AND Sex = ?',['A%',ord(sMale)])) do
        while aMale.FillOne do
            DoSomethingWith(aMale);
```

Or use one of the TSQLRecord.AutoComplete overloaded class methods:

```pascal
var aMale: TSQLBaby;

    with TSQLBaby.AutoComplete(aMale,Client,'Name LIKE ? AND Sex = ?',['A%',ord(sMale)]) do
        while aMale.FillOne do
            DoSomethingWith(aMale);
```

If you want your code to cross-compile with both Delphi and FPC, consider this expectation of the FPC compiler.

### 5.5. Objects relationship: cardinality

All previous code is fine if your application requires "flat" data. But most of the time, you'll need to define master/child relationship, perhaps over several levels. In data modeling, the *cardinality* of one data table with respect to another data table is a critical aspect of database design. Relationships between data tables define *cardinality* when explaining how each table links to another.

In the relational model, tables can have the following *cardinality*, i.e. can be related as any of:
- "One to one".
- "Many to one" (rev. "One to many");
- "Many to many" (or "has many").

Our mORMot framework handles all those kinds of *cardinality*.

#### 5.5.1. "One to one" or "One to many"

5.5.1.1. TSQLRecord published properties are IDs, not instance

In order to handle "One to one" or "One to many" relationship between tables (i.e. normalized Master/Detail in a classical RDBMS approach), you could define TSQLRecord published properties in the object definition.

For instance, you could declare classes as such:

```pascal
TSQLMFileInfo = class(TSQLRecord)
```
As stated by *TSQLRecord fields definition* (page 131), TSQLRecord published properties do not contain an instance of the TSQLRecord class. They will instead contain pointer(RowID), and will be stored as an INTEGER in the database.

So the main rule is to *never use directly such published properties*, as if they were regular class instance: otherwise you'll have an unexpected *access violation* error.

### 5.5.1.2. Transtyping IDs

When creating such records, use temporary instances for each detail object, as such:

```pascal
var
  One, Two: TSQLMyFileInfo;
  MyFile: TSQLMyFile;
begin
  One := TSQLMyFileInfo.Create;
  Two := TSQLMyFileInfo.Create;
  MyFile := TSQLMyFile.Create;
  try
    One.MyFileDate := ....
    One.MyFileSize := ...
    MyFile.FirstOne := TSQLMyFileInfo(MyDataBase.Add(One,True)); // add One and store ID in MyFile.FirstOne
    Two.MyFileDate := ....
    Two.MyFileSize := ...
    MyFile.SecondOne := TSQLMyFileInfo(MyDataBase.Add(Two,True)); // add Two and store ID in MyFile.SecondOne
  finally
    MyDataBase.Add(MyFile,true);
    MyFile.Free;
    Two.Free;
    One.Free;
  end;
end;
```

Note that those two assignments are the same:

```pascal
MyFile.FirstOne := TSQLMyFileInfo(MyDataBase.Add(One,True));
MyFile.FirstOne := pointer(MyDataBase.Add(One,True));
```

Or you may have added the One row first:

```pascal
MyDataBase.Add(One,true);
```

then assigned it to the MyFile record on one of the following expressions:

```pascal
MyFile.FirstOne := TSQLMyFileInfo(One.ID);
MyFile.FirstOne := pointer(One.ID);
MyFile.FirstOne := One.AsTSQLRecord;
```
The first two statements, using a class/pointer type cast will work only in 32-bit (since ID is an integer). Using TSQLRecord.AsTSQLRecord property will work on all platforms, including 64-bit, and is perhaps easier to deal with in your code.

When accessing the detail objects, you should not access directly to FirstOne or SecondOne properties (there are not class instances, but integer IDs), then use instead the TSQLRecord.Create(aClient: TSQLRest; aPublishedRecord: TSQLRecord: ForUpdate: boolean=false) overloaded constructor, as such:

```pascal
var One: TSQLMyFileInfo;
    MyFile: TSQLMyFile;
begin
    MyFile := TSQLMyFile.Create(Client,aMyFileID);
    try
        // here MyFile.FirstOne.MyFileDate will trigger an access violation
        One := TSQLMyFileInfo.Create(MyFile.FirstOne);
        try
            // here you can access One.MyFileDate or One.MyFileSize
        finally
            One.Free;
        end;
    finally
        MyFile.Free;
    end;
end;
```

Or with a with statement:

```pascal
var MyFile: TSQLMyFile;
begin
    MyFile := TSQLMyFile.Create(Client,aMyFileID);
    try
        // here MyFile.FirstOne.MyFileDate will trigger an access violation
        with TSQLMyFileInfo.Create(MyFile.FirstOne) do
        try
            // here you can access MyFileDate or MyFileSize
        finally
            Free;
        end;
    finally
        MyFile.Free;
    end;
end;
```

Mapping a TSQLRecord field into an integer ID is a bit difficult to learn at first. It was the only way we found out in order to define a "one to one" or "one to many" relationship within the class definition, without any property attribute features of the Delphi compiler (only introduced in newer versions). The main drawback is that the compiler won't be able to identify at compile time some potential GPF issues at run time. This is up to the developer to write correct code, when dealing with TSQLRecord properties. Using AsTSQLRecord property and overloaded TSQLRecord constructor will help a lot.

### 5.5.1.3. Automatic instantiation and JOINed query

Having to manage at hand all nested TSQLRecord instances can be annoying, and error-prone.

As an alternative, if you want to retrieve a whole TSQLRecord instance including its nested TSQLRecord published properties, you can use either of those two constructors:
- TSQLRecord.CreateJoined(aClient,aID);
- TSQLRecord.CreateAndFillPrepareJoined(), followed by a while FillOne do .... loop.

Both constructors:
- Will *auto-instantiate* all TSQLRecord published properties;
- Then the ORM core will retrieve all properties, included nested TSQLRecord via a SELECT .... LEFT JOIN ... statement;
- Then the nested TSQLRecord will be released at Destroy of the main instance (to avoid any unexpected memory leak).

So you can safely write:

```pascal
var MyFile: TSQLMyFile;
begin
  MyFile := TSQLMyFile.CreateJoined(Client,aMyFileID);
  try
    // here MyFile.FirstOne and MyFile.SecondOne are true instances
    // and have already retrieved from the database by the constructor
    // so you can safely access MyFile.FirstOne.MyFileDate or MyFile.SecondOne.MyFileSize here!
    finally
      MyFile.Free; // will release also MyFile.FirstOne and MyFile.SecondOne
    end;
  end;
end;
```

Note that this will work as expected when retrieving some data from the database, but, in the current implementation of the ORM, any `Update()` call will manage only the main TSQLRecord properties, and the nested TSQLRecord properties ID, not the nested properties values. For instance, in code above, `aClient.Update(MyFile)` will update the TSQLMyFile table, but won't reflect any modification to `MyFile.FirstOne` or `MyFile.SecondOne` properties. This limitation may be removed in the future - you may ask explicitly for this feature request.

### 5.5.2. "Has many" and "has many through"


In systems analysis, a many-to-many relationship is a type of cardinality that refers to the relationship between two entities (see also Entity-Relationship Model) A and B in which A may contain a parent row for which there are many children in B and vice versa. For instance, think of A as Authors, and B as Books. An Author can write several Books, and a Book can be written by several Authors. Because most database management systems only support one-to-many relationships, it is necessary to implement such relationships physically via a third and fourth junction table, say, AB with two one-to-many relationships A -> AB and B -> AB. In this case the logical primary key for AB is formed from the two foreign keys (i.e. copies of the primary keys of A and B).

From the record point of view, and to follow the ORM vocabulary (in Ruby on Rails, Python, or other ActiveRecord clones), we could speak of "has many" relationship. In the classic RDBMS implementation, a pivot table is created, containing two references to both related records. Additional information can be stored within this pivot table. It could be used, for instance, to store association time or corresponding permissions of the relationship. This is called a "has many through" relationship.

In fact, there are several families of ORM design, when implementing the "many to many" cardinality:
- Map collections into JOINed query from the ORM (i.e. pivot tables are abstracted from object lists or collections by the framework, to implement "has many" relationship, but you will have to define lazy loading and won't have "has many through" relationship at hand);
- Explicitly handle pivot tables as ORM classes, and provide methods to access to them (it will allow both "has many" and "has many through" relationship).
- Store collections within the ORM classes property (data sharding).

In the *mORMot framework*, we did not implement the 1st implementation pattern, but the 2nd and
3rd:
- You can map the DB with dedicated TSQLRecordMany classes, which allows some true pivot table to be available (that is the 2nd family), introducing true "has many through" cardinality;
- But for most applications, it sounds definitively more easy to use TCollection (of TPersistent classes) or dynamic arrays within one TSQLRecord class, and data sharding (i.e. the 3rd family).

Up to now, there is no explicit Lazy Loading feature in our ORM. There is no native handling of TSQLRecord collections or lists (as they do appear in the first family of ORMs). This could sound like a limitation, but it allows to manage exactly the data to be retrieved from the server in your code, and maintain bandwidth and memory use as low as possible. Use of a pivot table (via the TSQLRecordMany kind of records) allows tuned access to the data, and implements optimal lazy loading feature. Note that the only case when some TSQLRecord instances are automatically created by the ORM is for those TSQLRecordMany published properties.

5.5.2.1. Shared nothing architecture (or sharding)

5.5.2.1.1. Embedding all needed data within the record

Defining a pivot table is a classic and powerful use of relational database, and unleash its power (especially when linked data is huge).

But it is not easy nor natural to properly handle it, since it introduces some dependencies from the DB layer into the business model. For instance, it does introduce some additional requirements, like constraints / integrity checking and tables/classes inter-dependency.

Furthermore, in real life, we do not have such a separated storage, but we store all details within the main data. So for a Domain-Driven Design (page 99), which tries to map the real objects of its own domain, such a pivot table is breaking the business logic. With today's computer power, we can safely implement a centralized way of storing data into our data repository.


A shared nothing architecture (SN) is a distributed computing architecture in which each node is independent and self-sufficient, and there is no single point of contention across the system. People typically contrast SN with systems that keep a large amount of centrally-stored state information, whether in a database, an application server, or any other similar single point of contention.

As we stated in TSQLRecord fields definition (page 131), in our ORM, high-level types like dynamic arrays or TPersistent / TCollection properties are stored as BLOB or TEXT inside the main data row. There is no external linked table, no Master/Detail to maintain. In fact, each TSQLRecord instance content could be made self-contained in our ORM.

In particular, you may consider using our TDocVariant custom variant type (page 112) stored in a variant published property. It will allow to store any complex document, of nested objects or objects. They will be efficiently stored and transmitted as JSON.

When the server starts to have an increasing number of clients, such a data layout could be a major benefit. In fact, the so-called sharding, or horizontal partitioning of data, is a proven solution for web-scale databases, such as those in use by social networking sites. How does EBay or Facebook scale with so many users? Just by sharding.

A simple but very efficient sharding mechanism could therefore be implemented with our ORM. In-memory databases, or SQLite3 are good candidate for light speed data process. Even SQLite could scale very well in most cases, when properly used - see below (page 199).
Storing detailed data in BLOB or in TEXT as JSON could first sound a wrong idea. It does break one widely accepted principle of the RDBMS architecture. But even Google had to break this dogma. And when MySQL or any similar widely used databases try to implement sharding, it does need a lot of effort. Others, like the NoSQL MongoDB, are better candidates: they are not tight to the SQL/RDBMS flat scheme.

Finally, this implementation pattern fits much better with a Domain-Driven design. See below (page 595).

Therefore, on second thought, having at hand a shared nothing architecture could be a great advantage. Our ORM is already ready to break the table-oriented of SQL. Let us go one step further.

5.5.2.1.2. Nesting objects and arrays

The "has many" and "has many through" relationship we just described does follow the classic process of rows association in a relational database, using a pivot table. This does make sense if you have some DB background, but it is sometimes not worth it.

One drawback of this approach is that the data is split into several tables, and you should carefully take care of data integrity to ensure for instance that when you delete a record, all references to it are also deleted in the associated tables. Our ORM engine will take care of it, but could fail sometimes, especially if you play directly with the tables via SQL, instead of using high-level methods like FillMany* or DestGetJoined.

Another potential issue is that one business logical unit is split into several tables, therefore into several diverse TSQLRecord and TSQLRecordMany classes. From the ORM point of view, this could be confusing.

Starting with the revision 1.13 of the framework, dynamic arrays, TStrings and TCollection can be used as published properties in the TSQLRecord class definition. This won't be strong enough to implement all possible "Has many" architectures, but could be used in most case, when you need to add a list of records within a particular record, and when this list won't have to be referenced as a stand-alone table.

Dynamic arrays will be stored as BLOB fields in the database, retrieved with Base64 encoding in the JSON transmitted stream, then serialized using the TDynArray wrapper. Therefore, only Delphi clients will be able to use this field content: you'll loose the AJAX capability of the ORM, at the benefit of better integration with object pascal code. Some dedicated SQL functions have been added to the SQLite engine, like IntegerDynArrayContains, to search inside this BLOB field content from the WHERE clause of any search (see below (page 161)). Those functions are available from AJAX queries.

TPersistent / TStrings and TCollection / TObjectList will be stored as TEXT fields in the database, following the ObjectToJSON function format: you can even serialize any TObject class, via a previous call to the TJSONSerializer. RegisterCustomSerializer class method - see below (page 306) - or TObjectList list of instances, if they are previously registered by TJSONSerializer. RegisterClassForJSON - see below (page 309). This format contains only valid JSON arrays or objects: so it could be un-serialized via an AJAX application, for instance.

About this (trolling?) subject, and why/when you should use plain Delphi objects or arrays instead of classic Master/Detail DB relationship, please read "Objects, not tables" and "ORM is not DB" paragraphs below.

5.5.2.1.2.1. TDocVariant and variant fields
5.5.2.1.2.1. Schemaless storage via a variant

As we just wrote, a first-class candidate for data sharding in a TSQLRecord is our TDocVariant custom variant type (page 112).

You may define:

```pascal
TSQLRecordData = class(TSQLRecord)
private
  fName: RawUTF8;
  fData: variant;
public
  published
    property Name: RawUTF8 read fName write fName stored AS_UNIQUE;
    property Data: variant read fData write fData;
end;
```

Here, we defined two indexed keys, ready to access any data record:
- Via the ID: TID property defined at TSQLRecord level, which will map the SQLite3 RowID primary key;
- Via the Name: RawUTF8 property, which will was marked to be indexed by setting the "stored AS_UNIQUE" attribute.

Then, any kind of data may be stored in the Data: variant published property. In the database, it will be stored as JSON UTF-8 text, ready to be retrieved from any client, including AJAX / HTML5 applications. Delphi clients or servers will access those data via late-binding, from its TDocVariant instance.

You just reproduced the schema-less approach of the NoSQL database engines, in a few lines of code! Thanks to the mORMot's below (page 295) design, your applications are able to store any kind of document, and easily access to them via HTTP.

The documents stored in such a database can have varying sets of fields, with different types for each field. One could have the following objects in a single collection of our Data: variant rows:

```
{ name : "Joe", x : 3.3, y : [1,2,3] }
{ name : "Kate", x : "abc" }
{ q : 456 }
```

Of course, when using the database for real problems, the data does have a fairly consistent structure. Something like the following will be more common, e.g. for a table persisting student objects:

```
{ name : "Joe", age : 30, interests : "football" }
{ name : "Kate", age : 25 }
```

Generally, there is a direct analogy between this schema-less style and dynamically typed languages. Constructs such as those above are easy to represent in PHP, Python and Ruby. And, thanks to our TDocVariant late-binding magic, even our good Delphi is able to handle those structures in our code.

What we are trying to do here is make this mapping to the database natural, like:

```pascal
var aRec: TSQLRecordData;
aID: TID;
begin
  // initialization of one record
  aRec := TSQLRecordData.Create;
aRec.Name := 'Joe'; // one unique key
  aRec.data := _JSONFast('{name:"Joe",age:30}'); // create a TDocVariant
  // or we can use this overloaded constructor for simple fields
  aRec := TSQLRecordData.Create(['Joe',_ObjFast(['name','Joe','age',30])]);
  // now we can play with the data, e.g. via late-binding:
  writeln(aRec.Name); // will write 'Joe'
  writeln(aRec.Data); // will write '{"name":"Joe","age":30}' (auto-converted to JSON string)
  aRec.Data.age := aRec.Data.age+1; // one year older
```

One of the great benefits of these dynamic objects is that schema migrations become very easy. With a traditional RDBMS, releases of code might contain data migration scripts. Further, each release should have a reverse migration script in case a rollback is necessary. ALTER TABLE operations can be very slow and result in scheduled downtime.

With a schema-less organization of the data, 90% of the time adjustments to the database become transparent and automatic. For example, if we wish to add GPA to the student objects, we add the attribute, re-save, and all is well - if we look up an existing student and reference GPA, we just get back null. Further, if we roll back our code, the new GPA fields in the existing objects are unlikely to cause problems if our code was well written.

In fact, SQLite3 is so efficient about its indexes B-TREE storage, that such a structure may be used as a credible alternative to much heavier NoSQL engines, like MongoDB or CouchDB.

With the possibility to add some "regular" fields, e.g. plain numbers (like ahead-computed aggregation values), or text (like a summary or description field), you can still use any needed fast SQL query, without the complexity of map/reduce algorithm used by the NoSQL paradigm. You could even use the Full Text Search - FTS3/FTS4/FTS5, see below (page 215) - or RTREE extension advanced features of SQLite3 to perform your queries. Then, thanks to mORMot's ability to access any external database engine, you are able to perform a JOINed query of your schema-less data with some data stored e.g. in an Oracle, PostgreSQL or MS SQL enterprise database. Or switch later to a true MongoDB storage, in just one line of code - see below (page 287).

5.5.2.1.2.1.2. JSON operations from SQL code

As we stated, any variant field will be serialized as JSON, then stored as plain TEXT in the database. In order to make a complex query on the stored JSON, you could retrieve it in your end-user code, then use the corresponding TDocVariant instance to perform the search on its content. Of course, all this has a noticeable performance cost, especially when the data tend to grow.

The natural way of solving those performance issue is to add some "regular" RDBMS fields, with a proper index, then perform the requests on those fields. But sometimes, you may need to do some addition query, perhaps in conjunction with "regular" field lookup, on the JSON data stored itself.

In order to avoid the slowest conversion to the ORM client side, we defined some SQL functions, dedicated to JSON process.

The first is JsonGet(), and is able to extract any value from the TEXT field, mapping a variant:

| JsonGet(ArrColumn,0) | returns a property value by index, from a JSON array |
| JsonGet(ObjColumn,'PropName') | returns a property value by name, from a JSON object |
| JsonGet(ObjColumn,'Obj1.Obj2.Prop') | returns a property value by path, including nested JSON objects |
| JsonGet(ObjColumn,'Prop1,Prop2') | extract properties by name, from a JSON object |
JsonGet(ObjColumn,'Prop1,Obj1.Prop')  extract properties by name (including nested JSON objects), from a JSON object

JsonGet(ObjColumn,'Prop*')  extract properties by wildcard name, from a JSON object

JsonGet(ObjColumn,'Prop*,Obj1.P*')  extract properties by wildcard name (including nested JSON objects), from a JSON object

If no value does match, this function will return the SQL NULL. If the matching value is a simple JSON text or number, it will be returned as a TEXT, INTEGER or DOUBLE value, ready to be passed as a result column or any WHERE clause. If the returned value is a nested JSON object or array, it will be returned as TEXT, serialized as JSON; as a consequence, you may use it as the source of another JsonGet() function, or even able to gather the results via the CONCAT() aggregate function.

The comma-separated syntax allowed in the property name parameter (e.g. 'Prop1,Prop2,Prop3*'), will search for several properties at once in a single object, returning a JSON object of all matching values - e.g. '{"Prop2": "Value2", "Prop3": 123}' if the Prop1 property did not appear in the stored JSON object.

If you end the property name with a * character, it will return a JSON object, with all matching properties. Any nested object will have its property names be flattened as {"Obj1.Prop"...}, within the returned JSON object.

Note that the comma-separated syntax also allows such wildcard search, so that e.g.

```sql
JsonGet(ObjColumn,'owner') = {"login":"smith","id":123456} as TEXT
JsonGet(ObjColumn,'owner.login') = "smith" as TEXT
JsonGet(ObjColumn,'owner.id') = 123456 as INTEGER
JsonGet(ObjColumn,'owner.name') = NULL
JsonGet(ObjColumn,'owner.login,owner.id') = {"owner.login":"smith","owner.id":123456} as TEXT
JsonGet(ObjColumn,'owner.I*') = {"owner.id:123456} as TEXT
JsonGet(ObjColumn,'owner.*') = {"owner.login":"smith","owner.id":123456} as TEXT
JsonGet(ObjColumn,'unknown.*') = NULL
```

Another function, named JsonHas() is similar to JsonGet(), but will return TRUE or FALSE depending if the supplied property (specified by name or index) do exist. It may be faster to use JsonHas() than JsonGet() e.g. in a WHERE clause, when you do not want to process this property value, but only return data rows containing needed information.

```sql
JsonHas(ObjColumn,'owner') = true
JsonHas(ObjColumn,'owner.login') = true
JsonHas(ObjColumn,'owner.name') = false
JsonHas(ObjColumn,'owner.i*') = true
JsonHas(ObjColumn,'owner.n*') = false
```

Since the process will take place within the SQLite3 engine itself, and since they use a SAX-like fast approach (without any temporary memory allocation during its search), those JSON functions could be pretty efficient, and proudly compare to some dedicated NoSQL engines.

### 5.5.2.1.2.2. Dynamic arrays fields

#### 5.5.2.1.2.2.1. Dynamic arrays from Delphi Code

For instance, here is how the regression tests included in the framework define a TSQLRecord class with some additional *dynamic arrays* fields:

```delphi
TFV = packed record
  Major, Minor, Release, Build: integer;
  Main, Detailed: string;
end;
```
This TSQLRecordPeopleArray class inherits from TSQLRecordPeople, so it will add some new UTF8, Ints, Currency and FileVersion fields to this root class fields (FirstName, LastName, Data, YearOfBirth, YearOfDeath).

Some content is added to the PeopleArray table, with the following code:

```delphi
var V: TSQLRecordPeople;
VA: TSQLRecordPeopleArray;
FV: TFV;
(...)
V2.FillPrepare(Client, 'LastName=:(''Dali''):');
V2.FillOne do
begin
  VA.FillFrom(V2);
  // fast copy some content from TSQLRecordPeople
  inc(n);
  if n and 31=0 then begin
    VA.UTF8 := '';
    VA.DynArray('Ints').Add(n);
    Curr := n*0.01;
    VA.DynArray(2).Add(Curr);
    FV.Major := n;
    FV.Minor := n+2000;
    FV.Release := n+3000;
    FV.Build := n+4000;
    str(n,FV.Main);
    str(n+1000,FV.Detailed);
    VA.DynArray('FileVersion').Add(FV);
  end else
  str(n,VA.UTF8);
end
```

The n variable is used to follow the PeopleArray number, and will most of the time set its textual converted value in the UTF8 column, and once per 32 rows, will add one item to both VA and FV dynamic array fields.

We could have used normal access to VVA and FV dynamic arrays, as such:

```delphi
SetLength(VA.Ints,length(VA.Ints)+1);
VA.Ints[high(VA.Ints)] := n;
```

But the DynArray method is used instead, to allow direct access to the dynamic array via a TDynArray wrapper. Those two lines behave therefore the same as this code:

```delphi
VA.DynArray('Ints').Add(n);
```

Note that the DynArray method can be used via two overloaded set of parameters: either the field
name ('Ints'), or an index value, as was defined in the class definition. So we could have written:

```pascal
VA.DynArray(1).Add(n);
```

since the Ints published property has been defined as such:

```pascal
property Ints: TIntegerDynArray
index 1
read fInts write fInts;
```

Similarly, the following line will add a currency value to the Currency field:

```pascal
VA.DynArray(2).Add(Curr);
```

And a more complex TFV record is added to the FileVersion field *dynamic array* with just one line:

```pascal
VA.DynArray('FileVersion').Add(FV);
```

Of course, using the DynArray method is a bit slower than direct SetLength / Ints[] use. Using DynArray with an index should be also a bit faster than using DynArray with a textual field name (like 'Ints'), with the benefit of perhaps less keyboard errors at typing the property name. But if you need to fast add a lot of items to a *dynamic array*, you could use a custom TDynArray wrapper with an associated external Count value, or direct access to its content (like SetLength + Ints[])

Then the FillPrepare / FillOne loop ends with the following line:

```pascal
Check(Client.Add(VA,true)=n);
end;
```

This will add the VA fields content into the database, creating a new row in the PeopleArray table, with an ID following the value of the n variable. All *dynamic array* fields will be serialized as BLOB into the database table.

### 5.5.2.1.2.2. Dynamic arrays from SQL code

In order to access the BLOB content of the dynamic arrays directly from SQL statements, some new SQL functions have been defined in TSQLDataBase, named after their native simple types:

- ByteDynArrayContains(BlobField,I64);
- WordDynArrayContains(BlobField,I64);
- IntegerDynArrayContains(BlobField,I64);
- CardinalDynArrayContains(BlobField,I64);
- CurrencyDynArrayContains(BlobField,I64) - in this case, I64 is not the currency value directly converted into an Int64 value (i.e. not Int64(aCurrency)), but the binary mapping of the currency value, i.e. aCurrency*10000 or PInt64(@aCurrency)^;
- Int64DynArrayContains(BlobField,I64);
- RawUTF8DynArrayContainsCase(BlobField,'Text');
- RawUTF8DynArrayContainsNoCase(BlobField,'Text').

Those functions allow direct access to the BLOB content like this:

```pascal
for i := 1 to n shr 5 do
begin
  k := i shl 5;
  aClient.OneFieldValues(TSQLRecordPeopleArray,'ID',
    FormatUTF8('IntegerDynArrayContains(Ints,?)',[],[k]),IDS);
  Check(length(IDs)=n+1-32*i);
  for j := 0 to high(IDs) do
    Check(IDs[j]=k+j);
end;
```

In the above code, the WHERE clause of the OneFieldValues method will use the dedicated IntegerDynArrayContains SQL function to retrieve all records containing the specified integer
value k in its Ints BLOB column. With such a function, all the process is performed Server-side, with no slow data transmission nor JSON/Base64 serialization.

For instance, using such a SQL function, you are able to store multiple TSQLRecord.ID field values into one TIntegerDynArray property column, and have direct search ability inside the SQL statement. This could be a very handy way of implementing "one to many" or "many to many" relationship, without the need of a pivot table.

Those functions were implemented to be very efficient for speed. They won’t create any temporary dynamic array during the search, but will access directly to the BLOB raw memory content, as returned by the SQLite engine. The RawUTF8DynArrayContainsCase / RawUTF8DynArrayContainsNoCase functions also will search directly inside the BLOB. With huge number of requests, this could be slower than using a TSQLRecordMany pivot table, since the search won’t use any index, and will have to read all BLOB field during the request. But, in practice, those functions behave nicely with a relative small amount of data (up to about 50,000 rows). Don't forget that BLOB column access are very optimized in SQLite3.

For more complex dynamic array content handling, you’ll have either to create your own SQL function using the TSQLDataBase. RegisterSQLFunction method and an associated TSQLDataBaseSQLFunction class, or via a dedicated Service or a stored procedure - see below (page 363) on how to implement it.

5.5.2.1.2.3. TPersistent/TCollection fields

For instance, here is the way regression tests included in the framework define a TSQLRecord class with some additional TPersistent, TCollection or TRawUTF8List fields (TRawUTF8List is just a TStringList-like component, dedicated to handle RawUTF8 kind of string):

```pascal
TSQLRecordPeopleObject = class(TSQLRecordPeople)
  private
    fPersistent: TCollTst;
    fUTF8: TRawUTF8List;
  public
    constructor Create; override;
    destructor Destroy; override;
  published
    property UTF8: TRawUTF8List read fUTF8;
    property Persistent: TCollTst read fPersistent;
  end;
```

In order to avoid any memory leak or access violation, it is mandatory to initialize then release all internal property instances in the overridden constructor and destructor of the class:

```pascal
constructor TSQLRecordPeopleObject.Create;
begin
  inherited;
  fPersistent := TCollTst.Create;
  fUTF8 := TRawUTF8List.Create;
end;
```

```pascal
destructor TSQLRecordPeopleObject.Destroy;
begin
  inherited;
  FreeAndNil(fPersistent);
  FreeAndNil(fUTF8);
end;
```

Here is how the regression tests are performed:

```pascal
var VO: TSQLRecordPeopleObject;
(...)```
If Client.TransactionBegin(TSQLRecordPeopleObject) then
try
  V2.FillPrepare(Client, 'LastName=?', ['Morse']);
  n := 0;
  while V2.FillOne do
  begin
    VO.FillFrom(V2); // fast copy some content from TSQLRecordPeople
    inc(n);
    VO.Persistent.One.Color := n+100;
    VO.Persistent.One.Length := n;
    VO.Persistent.One.Name := Int32ToUtf8(n);
    if n and 31=0 then
      begin
        VO.UTF8.Add(VO.Persistent.One.Name);
        with VO.Persistent.Coll.Add do
        begin
          Color := n+1000;
          Length := n*2;
          Name := Int32ToUtf8(n*3);
        end;
      end;
      Check(Client.Add(VO, true)=n);
  end;
  Client.Commit;
except
  Client.Rollback; // in case of error
end;

This will add 1000 rows to the PeopleObject table.

First of all, the adding is nested inside a transaction call, to speed up SQL INSERT statements, via
TransactionBegin and Commit methods. Please note that the TransactionBegin method returns a
boolean value, and should be checked in a multi-threaded or Client-Server environment (in this part
of the test suit, content is accessed in the same thread, so checking the result is not mandatory, but
shown here for accuracy). In the current implementation of the framework, transactions should not be
nested. The typical transaction usage should be the following:

If Client.TransactionBegin(TSQLRecordPeopleObject) then
try
  //.... modify the database content, raise exceptions on error
  Client.Commit;
except
  Client.Rollback; // in case of error
end;

In a Client-Server environment with multiple Clients connected at the same time, you can use the
dedicated TSQLRestClientURI.TransactionBeginRetry method:

If Client.TransactionBeginRetry(TSQLRecordPeopleObject, 20) then
...

Note that the transactions are handled according to the corresponding client session: the client should
make the transaction block as short as possible (e.g. using a batch command), since any write attempt
by other clients will wait for the transaction to be released (with either a commit or rollback).

The fields inherited from the TSQLRecord class are retrieved via FillPrepare / FillOne method calls,
for columns with the LastName matching 'Morse'. One TPersistent property instance values are set
(VO.Persistent.One), then, for every 32 rows, a new item is added to the VO.Persistent.Coll
collection.

Here is the data sent for instance to the Server, when the item with ID=32 is added:

{"FirstName":"Samuel Finley Breese31",
 "LastName":"Morse",
 "YearOfBirth":1791,"}
"YearOfDeath":1872,
"UTF8":["32"],
"Persistent":{"One":{"Color":132,"Length":32,"Name":"32"},"Coll":[{"Color":1032,"Length":64,"Name":"96"}]}
}

Up to revision 1.15 of the framework, the transmitted JSON content was not a true JSON object, but sent as RawUTF8 TEXT values (i.e. every double-quote ("" character is escaped as - e.g. "UTF8":"[32"]". Starting with revision 1.16 of the framework, the transmitted data is a true JSON object, to allow better integration with an AJAX client. That is, UTF8 field is transmitted as a valid JSON array of string, and Persistent as a valid JSON object with nested objects and arrays.

When all 1000 rows were added to the database file, the following loop is called once with direct connection to the DB engine, once with a remote client connection (with all available connection protocols):

```delphi
for i := 1 to n do
begin
  VO.ClearProperties;
  Client.Retrieve(i,VO);
  Check(VO.ID=i);
  Check(VO.LastName='Morse');
  Check(VO.UTF8.Count=i shr 5);
  for j := 0 to VO.UTF8.Count-1 do
    Check(GetInteger(pointer(VO.UTF8[j]))=(j+1) shl 5);
  Check(VO.Persistent.One.Length=i);
  Check(VO.Persistent.One.Color=i+100);
  Check(GetInteger(pointer(VO.Persistent.One.Name))=i);
  Check(VO.Persistent.Coll.Count=i shr 5);
  for j := 0 to VO.Persistent.Coll.Count-1 do
    with VO.Persistent.Coll[j] do
    begin
      k := (j+1) shl 5;
      Check(Color=k+1000);
      Check(Length=k*2);
      Check(GetInteger(pointer(Name))=k*3);
    end;
end;
```

All the magic is made in the Client.Retrieve(i,VO) method. Data is retrieved from the database as TEXT values, then un-serialized from JSON arrays or objects into the internal TRawUTF8List and TPersistent instances.

When the ID=33 row is retrieved, the following JSON content is received from the server:

```json
{"ID":33,
 "FirstName":"Samuel Finley Breese32",
 "LastName":"Morse",
 "YearOfBirth":1791,
 "YearOfDeath":1872,
 "UTF8":"["32"]",
 "Persistent":{"One":{"Color":133,"Length":33,"Name":"33"},"Coll":[{"Color":1032,"Length":64,"Name":"96"}]}}
```

In contradiction with POST content, this defines no valid nested JSON objects nor arrays, but UTF8 and Persistent fields transmitted as JSON strings. This is a known limitation of the framework, due to the fact that it is much faster to retrieve directly the text from the database than process for this operation. For an AJAX application, this won't be difficult to use a temporary string property, and evaluate the JSON content from it, in order to replace the property with a corresponding object content. Implementation may change in the future.

5.5.2.1.2.4. Any TObject, including TObjectList
Not only TPersistent, TCollection and TSQLRecord types can be serialized by writing all published properties. The ORM core of mORMot uses ObjectToJSON() and JSONToObject() (aka TJSONObjectizer.WriteObject) functions to process proper JSON serialization.

You have two methods to register JSON serialization for any kind of class:
- Custom serialization via read and write callbacks - see TJSONObjectizer. RegisterCustomSerializer below (page 306);
- TObjectList instances, after a proper call to TJSONObjectizer. RegisterClassForJSON below (page 309).

In the database, such kind of objects will be stored as TEXT (serialized as JSON), and transmitted as regular JSON objects or arrays when working in Client-Server mode.

5.5.2.1.2.5. Sharding on NoSQL engines
This "Shared nothing architecture" matches perfectly with the NoSQL and Object-Document Mapping (ODM) (page 96) design.

In fact, mORMot's integration with MongoDB has been optimized so that any of those high-level properties (like dynamic arrays, variants and TDocVariant, or any class) will be stored as BSON documents on the MongoDB server.

If those types are able to be serialized as JSON - which is the case for simple types, variants and for any dynamic array / record custom types - see below (page 303), then the mORMotDB.pas unit will store this data as BSON objects or arrays on the server side, and not as BLOB or JSON text (as with SQL back-ends). You will be able to query by name any nested sub-document or sub-array, in the MongoDB collection.

As such, data sharing with mORMot will benefit of RDBMS back-end, as a reliable and proven solution, but also of the latest NoSQL technology.

5.5.2.2. ORM implementation via pivot table
Data sharding just feels natural, from the ORM point of view.

But defining a pivot table is a classic and powerful use of relational database, and will unleash its power:
- When data is huge, you can query only for the needed data, without having to load the whole content (it is something similar to lazy loading in ORM terminology);
- In a master/detail data model, sometimes it can be handy to access directly to the detail records, e.g. for data consolidation;
- And, last but not least, the pivot table is the natural way of storing data associated with "has many through" relationship (e.g. association time or corresponding permissions).

5.5.2.2.1. Introducing TSQLRecordMany
A dedicated class, inheriting from the standard TSQLRecord class (which is the base of all objects stored in our ORM), has been created, named TSQLRecordMany. This table will turn the "many to many" relationship into two "one to many" relationships pointing in opposite directions. It shall contain at least two TSQLRecord (i.e. INTEGER) published properties, named "Source" and "Dest" (name is mandatory, because the ORM will share for exact matches). The first pointing to the source record (the one with a TSQLRecordMany published property) and the second to the destination record.

For instance:
TSQLDest = class(TSQLRecord);
TSQLSource = class;
TSQLDestPivot = class(TSQLRecordMany)
private
  fSource: TSQLSource;
  fDest: TSQLDest;
  fTime: TDateTime;
published
  property Source: TSQLSource read fSource; // map Source column
  property Dest: TSQLDest read fDest; // map Dest column
  property AssociationTime: TDateTime read fTime write fTime;
end;
TSQLSource = class(TSQLRecordSigned)
private
  fDestList: TSQLDestPivot;
published
  property SignatureTime;
  property Signature;
  property DestList: TSQLDestPivot read fDestList;
end;
TSQLDest = class(TSQLRecordSigned)
published
  property SignatureTime;
  property Signature;
end;

When a TSQLRecordMany published property exists in a TSQLRecord, it is initialized automatically during TSQLRecord.Create constructor execution into a real class instance. Note that the default behavior for a TSQLRecord published property is to contain an INTEGER value which is the ID of the corresponding record - creating a "one to one" or "many to one" relationship. But TSQLRecordMany is a special case. So don't be confused! :)

This TSQLRecordMany instance is indeed available to access directly the pivot table records, via FillMany then FillRow, FillOne and FillRewind methods to loop through records, or FillManyFromDest / DestGetJoined for most advanced usage.

Here is how the regression tests are written in the SynSelfTests unit:

procedure TestMany(aClient: TSQLRestClient);
var
  MS: TSQLSource;
  MD, MD2: TSQLDest;
  i: integer;
  sID, dID: array[1..100] of Integer;
  res: TIntegerDynArray;
begin
  MS := TSQLSource.Create;
  MD := TSQLDest.Create;
  try
    MD.fSignatureTime := TimeLogNow;
    MS.fSignatureTime := MD.fSignatureTime;
    Check(MS.DestList<>nil);
    Check(MS.DestList.InheritsFrom(TSQLRecordMany));
    aClient.TransactionBegin(TSQLSource); // faster process
    for i := 1 to high(dID) do
      begin
        MD.fSignature := FormatUTF8('%% %', [aClient.ClassName, i]);
        dID[i] := aClient.Add(MD, true);
        Check(dID[i]>0);
      end;
  except
    on E: EdoSQL do
      Check(E.Message = 'CONCURRENT ACCESS');
  end;
end;

This code will create two TSQLSource / TSQLDest instances, then will begin a transaction (for faster database engine process, since there will be multiple records added at once). Note that during TSQLSource.Create execution, the presence of a TSQLRecordMany field is detected, and the DestList property is filled with an instance of TSQLDestPivot. This DestList property is therefore able to be directly used via the "has-many" dedicated methods, like ManyAdd.
This will just add some rows to the Dest table.

```
for i := 1 to high(sID) do begin
  MS.fSignature := FormatUTF8('% %',[aClient.ClassName,i]);
  sID[i] := aClient.Add(MS,True);
  Check(sID[i]>0);
  MS.DestList.AssociationTime := i;
  Check(MS.DestList.ManyAdd(aClient,sID[i],dID[i])); // associate both lists
  Check(not MS.DestList.ManyAdd(aClient,sID[i],dID[i],true)); // no dup
end;
aClient.Commit;
```

This will create some Source rows, and will call the ManyAdd method of the auto-created DestList instance to associate a Dest item to the Source item. The AssociationTime field of the DestList instance is set, to implement a "has many through" relationship.

Then the transaction is committed to the database.

```
for i := 1 to high(dID) do begin
  Check(MS.DestList.SourceGet(aClient,dID[i],res));
  if not Check(length(res)=1) then
    Check(res[0]=sID[i]);
  Check(MS.DestList.ManySelect(aClient,sID[i],dID[i]));
  Check(MS.DestList.AssociationTime=i);
end;
```

This code will validate the association of Source and Dest tables, using the dedicated SourceGet method to retrieve all Source items IDs associated to the specified Dest ID, i.e. one item, matching the sID[] values. It will also check for the AssociationTime as set for the "has many through" relationship.

```
for i := 1 to high(sID) do begin
  Check(MS.DestList.DestGet(aClient,sID[i],res));
  if Check(length(res)=1) then
    continue; // avoid GPF
  Check(res[0]=dID[i]);
end;
```

The DestGet method retrieves all Dest items IDs associated to the specified Source ID, i.e. one item, matching the dID[] values.

```
Check(MS.DestList.FillMany(aClient,sID[i])=1);
```

This will fill prepare the DestList instance with all pivot table instances matching the specified Source ID. It should return only one item.

```
Check(MS.DestList.FillOne);
Check(Integer(MS.DestList.Source)=sID[i]);
Check(Integer(MS.DestList.Dest)=dID[i]);
Check(MS.DestList.AssociationTime=i);
Check(not MS.DestList.FillOne);
```

Those lines will fill the first (and unique) prepared item, and check that Source, Dest and AssociationTime properties match the expected values. Then the next call to FillOne should fail, since only one prepared row is expected for this Source ID.

```
Check(MS.DestList.DestGetJoined(aClient,'',sID[i],res));
if not Check(length(res)=1) then
  Check(res[0]=dID[i]);
```

This will retrieve all Dest items IDs associated to the specified Source ID, with no additional WHERE condition.

```
Check(MS.DestList.DestGetJoined(aClient,'Dest.SignatureTime=:(0):',sID[i],res));
```
Check(length(res)=0);

This will retrieve all Dest items IDs associated to the specified Source ID, with an additional always invalid WHERE condition. It should always return no item in the res array, since SignatureTime is never equal to 0.

Check(MS.DestList.DestGetJoined(aClient,
    FormatUTF8('Dest.SignatureTime=?',[],[MD.SignatureTime]),sID[i],res));
if Check(length(res)=1) then
    continue; // avoid GPF
    Check(res[0]=dID[i]);

This will retrieve all Dest items IDs associated to the specified Source ID, with an additional WHERE condition, matching the expected value. It should therefore return one item.

Note the call of the global FormatUTF8() function to get the WHERE clause. You may have written instead:

Check(MS.DestList.DestGetJoined(aClient,
    'Dest.SignatureTime=('+Int64ToUTF8(MD.SignatureTime)+')',sID[i],res));

But in this case, using manual inlined :(...) values is less convenient than the '?' calling convention, especially for string (RawUTF8) values.

MD2 := MS.DestList.DestGetJoined(aClient,
    FormatUTF8('Dest.SignatureTime=?',[],[MD.SignatureTime]),sID[i]) as TSQLADest;
if Check(MD2<>nil) then
    try
        Check(MD2.FillOne);
        Check(MD2.ID=dID[i]);
        Check(MD2.Signature=FormatUTF8('% %',[aClient.ClassName,i]));
    finally
        MD2.Free;
    end;
end;

This overloaded DestGetJoined method will return into MD2 a TSQLADest instance, prepared with all the Dest record content associated to the specified Source ID, with an additional WHERE condition, matching the expected value. Then FillOne will retrieve the first (and unique) matching Dest record, and checks for its values.

aClient.TransactionBegin(TSQLADests); // faster process
for i := 1 to high(sID) shr 2 do
    Check(MS.DestList.ManyDelete(aClient,sID[i*4],dID[i*4]));
aClient.Commit;
for i := 1 to high(sID) do
    if i and 3<>0 then
        begin
            Check(MS.DestList MANYSelect(aClient,sID[i],dID[i]));
            Check(MS.DestList.AssociationTime=i);
        end else
        Check(not MS.DestList MANYSelect(aClient,sID[i],dID[i]));

This code will delete one association per four, and ensure that ManySelect will retrieve only expected associations.

finally
    MD.Free;
    MS.Free;
end;

This will release associated memory, and also the instance of TSQLDestPivot created in the DestList property.
5.5.2.2.2. Automatic JOIN query

All those methods (ManySelect, DestGetJoined...) are used to retrieve the relations between tables from the pivot table point of view. This saves bandwidth, and can be used in most simple cases, but it is not the only way to perform requests on many-to-many relationships. And you may have several TSQLRecordMany instances in the same main record - in this case, those methods won't help you.

It is very common, in the SQL world, to create a JOINed request at the main "Source" table level, and combine records from two or more tables in a database. It creates a set that can be saved as a table or used as is. A JOIN is a means for combining fields from two or more tables by using values common to each. Writing such JOINed statements is not so easy by hand, especially because you'll have to work with several tables, and have to specify the exact fields to be retrieved; if you have several pivot tables, it may start to be a nightmare. Let's see how our ORM will handle it.

A dedicated FillPrepareMany method has been added to the TSQLRecord class, in conjunction with a new constructor named CreateAndFillPrepareMany. This particular method will:

- Instantiate all Dest properties of each TSQLRecordMany instances - so that the JOINed request will be able to populate directly those values;
- Create the appropriate SELECT statement, with an optional WHERE clause.

Here is the test included in our regression suite, working with the same database:

```delphi
Check(MS.FillPrepareMany(aClient, 'DestList.Dest.SignatureTime<>% and id>=? and DestList.AssociationTime<>0 ' + 'and SignatureTime=DestList.Dest.SignatureTime ' + 'and DestList.Dest.Signature<>"DestList.AssociationTime"',[0],[sID[1]]));
```

Of course, the only useful parameter here is id>=? which is used to retrieve the just added relationships in the pivot table. All other conditions will always be true, but it will help testing the generated SQL.

Our mORMot will generate the following SQL statement:

```sql
select A.ID AID,A.SignatureTime A00,A.Signature A01,
    B.ID BID,B.AssociationTime B02,
    C.ID CID,C.SignatureTime C00,C.Signature C01
from ASource A,ADests B,ADest C
where B.Source=A.ID and B.Dest=C.ID
    and (C.SignatureTime<>0 and A.id>=:(1): and B.AssociationTime<>0
        and A.SignatureTime=c.SignatureTime and C.Signature="DestList.AssociationTime")
```

You can notice the following:

- All declared TSQLRecordMany instances (renamed B in our case) are included in the statement, with all corresponding Dest instances (renamed as C);
- Fields are aliased with short unique identifiers (AID, A01, BID, B02...), for all simple properties of every classes;
- The JOIN clause is created (B.Source=A.ID and B.Dest=C.ID);
- Our manual WHERE clause has been translated into proper SQL, including the table internal aliases (A,B,C) - in fact, DestList.Dest has been replaced by C, the main ID property has been declared properly as A.ID, and the "DestList.AssociationTime" text remained untouched, because it was bounded with quotes.

That is, our ORM did make all the dirty work for you! You can use Delphi-level conditions in your query, and the engine will transparently convert them into a valid SQL statement. Benefit of this will become clear in case of multiple pivot tables, which are likely to occur in real-world applications.

After the statement has been prepared, you can use the standard FillOne method to loop through all
returned rows of data, and access to the JOINed columns within the Delphi objects instances:

```delphi
Check(MS.FillTable.RowCount=length(sID));
for i := 1 to high(sID) do begin
  MS.FillOne;
  Check(MS.fID=sID[i]);
  Check(MS.SignatureTime=MD.fSignatureTime);
  Check(MS.DestList.AssociationTime=i);
  Check(MS.DestList.Dest.fID=dID[i]);
  Check(MS.DestList.Dest.SignatureTime=MD.fSignatureTime);
  Check(MS.DestList.Dest.Signature=FormatUTF8('% %',[aClient ClassName,i]));
end;
MS.FillClose;
```

Note that in our case, an explicit call to FillClose has been added in order to release all Dest instances created in FillPrepareMany. This call is not mandatory if you call MS.Free directly, but it is required if the same MS instance is about to use some regular many-to-many methods, like MS.DestList.ManySelect() - it will prevent any GPF exception to occur with code expecting the Dest property not to be an instance, but a pointer(DestID) value.
5.6. ORM Data Model

5.6.1. Creating an ORM Model

The TSQLModel class centralizes all TSQLRecord inherited classes used by an application, both
database-related and business-logic related.

In order to follow the MVC pattern, the TSQLModel instance is to be used when you have to deal at
table level. For instance, do not try to use low-level TSQLDataBase.GetTableNames or
TSQLDataBase.GetFieldNames methods in your code. In fact, the tables declared in the Model may
not be available in the SQLite3 database schema, but may have been defined as
TSQLRestStorageInMemory instance via the TSQLRestServer.StaticDataCreate method, or being
external tables - see below (page 239). You could even have a mORMot server running without any
SQLite3 engine at all, but pure in-memory tables!

Each TSQLModel instance is in fact associated with a TSQLRest instance. An Owner property gives
access to the current running client or server TSQLRest instance associated with this model.

By design, models are used on both Client and Server sides. It is therefore a good practice to use a
common unit to define all TSQLRecord types, and have a common function to create the related
TSQLModel class.

For instance, here is the corresponding function as defined in the first samples available in the source
code repository (unit SampleData.pas):

```pascal
function CreateSampleModel: TSQLModel;
begin
  result := TSQLModel.Create([TSQLSampleRecord]);
end;
```

For a more complex model including link to User Interface, see below (page 504).

5.6.2. Several Models

In practice, a same TSQLRecord can be used in several models: this is typically the case for
TSQLAuthUser tables, or if client and server instances are running in the same process. So, for
accessing the model properties, you have two structures available:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLModelRecordProperties</td>
<td>Model-specific ORM parameters, like dedicated SQL auto-generation and external DB settings. Access these instances from TSQLModel.TableProps[] array</td>
</tr>
<tr>
<td>TSQLRecordProperties</td>
<td>Low-level table properties, as retrieved from below (page 503) by the ORM kernel of mORMot. Access these instances from TSQLModel.TableProps[].Props</td>
</tr>
</tbody>
</table>

So you may use code like this:

```pascal
var i: integer;
  ModelProps: TSQLModelRecordProperties;
  Props: TSQLRecordProperties;
begin
  ...
  for i := 0 to high(Model.TableProps) do begin
```

...
ModelProps := Model.TableProps[i];
   // now you can access ModelProps.ExternalDB.TableName ...
Props := ModelProps.Props;
   // now you can use Props.SQLTableName or Props.Fields[]
end;
end;

5.6.3. Filtering and Validating

According to the n-Tier architecture - see Multi-tier architecture (page 88) - data filtering and validation should be implemented in the business logic, not in the User Interface.

If you were used to develop RAD database application using Delphi, you may have to change a bit your habits here. Data filtering and validation should be implemented not in the User Interface, but in pure Delphi code.

In order to make this easy, a dedicated set of classes are available in the SynCommons.pas unit, and allow to define both filtering (transformation) and validation. They all will be children of any of those both classes:

```
TSynValidate
```
```
TSynFilter
```
```
TSynFilterOrValidate
```

Filtering and Validation classes hierarchy

TSQLRecord field content filtering is handled in the TSQLRecord. Filter virtual method, or via some TSQLFilter classes. They will transform the object fields following some rules, e.g. forcing uppercase/lowercase, or trimming text spaces.

TSQLRecord field content validation is handled in the TSQLRecord. Validate virtual method, or via some TSQLValidate classes. Here the object fields will be checked against a set of rules, and report any invalid content.

Some "standard" classes are already defined in the SynCommons.pas and mORMot.pas units, covering most common usage:
You have powerful validation classes for IP Address, Email (with TLD+domain name), simple regex pattern, textual validation, strong password validation...

Note that some database-related validation is existing, like TSynValidateUniqueField which inherits from TSynValidateRest.

Of course, the mORMotUIEdit unit handles TSQLRecord automated filtering (using TSQLFilter classes) and validation (via the TSQLValidate classes).

The field validation process is run in TSQLRecord. Validate and not in mORMotUIEdit itself (to have a better multi-tier architecture).

To initialize it, you can add some filters/validators to your TSQLModel creation function:

```pascal
function CreateFileModel(Owner: TSQLRest): TSQLModel;
begin
result := TSQLModel.Create(Owner,
@FileTabs,length(FileTabs),sizeof(FileTabs[0]),[],
TypeInfo(TFileAction),TypeInfo(TFileEvent),'synfile');
TSQLFile.AddFilterOrValidate('Name',TSQLFilterLowerCase);
TSQLUser.AddFilterOrValidate('Email',TSQLFilterLowerCase);
end;
```

As an alternative, you can override the following method:

```pascal
TSQLRecordAuthInfo = class(TSQLRecord)
protected
```

Default filters and Validation classes hierarchy
class procedure InternalDefineModel(Props: TSQLRecordProperties); override;
...

class procedure TSQLRecordAuthInfo.InternalDefineModel(
  Props: TSQLRecordProperties);
begin
  AddFilterNotVoidText([''Logon'',''HashedPassword'']);
end;

It does make sense to define this behavior within the TSQLRecord definition, so that it will be shared by all models.

If you want to perform some text field length validation or filter at ORM level, you may use TSQLRecordProperties's SetMaxLengthValidatorForTextFields() or SetMaxLengthFilterForTextFields() method, or at model level:

function CreateModel: TSQLModel;
begin
  result := TSQLModel.Create([TSQLMyRecord1,TSQLMyRecord2]);
  result.SetMaxLengthValidatorForAllTextFields(true); // "index n" is in UTF-8 bytes
end;

In order to perform the filtering (transformation) of some content, you'll have to call the aRecord.Filter() method, and aRecord.Validate() to test for valid content.

For instance, this is how mORMotUIEdit.pas unit filters and validates the user interface input:

procedure TRecordEditForm.BtnSaveClick(Sender: TObject);
{
  ...
  // perform all registered filtering
  Rec.Filter(ModifiedFields);
  // perform content validation
  FieldIndex := -1;
 ErrMsg := Rec.Validate(Client,ModifiedFields,@FieldIndex);
  if ErrMsg<>'' then begin
    // invalid field content -> show message, focus component and abort saving
    if cardinal(FieldIndex)<cardinal(length(fFieldComponents)) then begin
      C := fFieldComponents[FieldIndex];
      C.SetFocus;
      Application.ProcessMessages;
      ShowMessage(ErrMsg,format(sInvalidFieldN,[fFieldCaption[FieldIndex]]),true);
    end else
    ShowMessage(ErrMsg,format(sInvalidFieldN,['?']),true);
  end else
  // close window on success
  ModalResult := mrOk;
end;

It is up to your code to filter and validate the record content. By default, the mORMot CRUD operations won't call the registered filters or validators.

5.7. ORM Cache

Here is the definition of "cache", as stated by Wikipedia:

In computer engineering, a cache is a component that transparently stores data so that future requests for that data can be served faster. The data that is stored within a cache might be values that have been computed earlier or duplicates of original values that are stored elsewhere. If requested data is contained in the cache (cache hit), this request can be served by simply reading the cache, which is comparatively faster. Otherwise (cache miss), the data has to be recomputed or fetched from its original storage location, which is comparatively slower. Hence, the greater the number of requests that can be served from the cache, the faster the overall system performance becomes.
To be cost efficient and to enable an efficient use of data, caches are relatively small. Nevertheless, caches have proven themselves in many areas of computing because access patterns in typical computer applications have locality of reference. References exhibit temporal locality if data is requested again that has been recently requested already. References exhibit spatial locality if data is requested that is physically stored close to data that has been requested already.

In our ORM framework, since performance was one of our goals since the beginning, cache has been implemented at four levels:

- Statement cache for implementing SQL prepared statements, and parameters bound on the fly - see Query parameters (page 145) and below (page 213) - note that this cache is available not only for the SQLite3 database engine, but also for any external engine - see below (page 239);
- Global JSON result cache at the database level, which is flushed globally on any INSERT / UPDATE - see below (page 317);
- Tuned record cache at the CRUD/RESTful level for specified tables or records on the server side - see below (page 359);
- Tuned record cache at the CRUD/RESTful level for specified tables or records on the client side - see below (page 359).

Thanks to those specific caching abilities, our framework is able to minimize the number of client-server requests, therefore spare bandwidth and network access, and scales well in a concurrent rich client access architecture. In such perspective, a Client-Server ORM does make sense, and is of huge benefit in comparison to a basic ORM used only for data persistence and automated SQL generation.
5.8. Calculated fields

It is often useful to handle some calculated fields. That is, having some field values computed when you set another field value. For instance, if you set an error code from an enumeration (stored in an INTEGER field), you may want the corresponding text (to be stored on a TEXT field). Or you may want a total amount to be computed automatically from some detailed records.

This should not be done on the Server side. In fact, the framework expects the transmitted JSON transmitted from client to be set directly to the database layer, as stated by this code from the mORMotSQLite3 unit:

```pascal
function TSQLRestServerDB.EngineUpdate(Table: TSQLRecordClass; ID: TID; const SentData: RawUTF8): boolean;
begin
if (self=nil) or (Table=nil) or (ID<=0) then
result := false
else
begin
// this SQL statement use :(inlined params): for all values
result := ExecuteFmt('UPDATE % SET % WHERE RowID=:(%):;',
[Table.RecordProps.SQLTableName,GetJSONObjectAsSQL(SentData,true,true),ID]);
if Assigned(OnUpdateEvent) then
OnUpdateEvent(self,seUpdate,Table,ID);
end;
end;
end;
```

The direct conversion from the received JSON content into the SQL UPDATE statement values is performed very quickly via the GetJSONObjectAsSQL procedure. It won't use any intermediary TSQLRecord, so there will be no server-side field calculation possible.

Record-level calculated fields should be done on the Client side, using some setters.

There are at least three ways of updating field values before sending to the server:
- Either by using some dedicated setters method for TSQLRecord properties;
- Or by overriding the ComputeFieldsBeforeWrite virtual method of TSQLRecord.
- If the computed fields need a more complex implementation (e.g. if some properties of another record should be modified), a dedicated RESTful service should be implemented - see below (page 372).

5.8.1. Setter for TSQLRecord

For instance, here we define a new table named `INVOICE`, with only two fields. A dynamic array containing the invoice details, then a field with the total amount. The dynamic array property will be stored as BLOB into the database, and no additional Master/Detail table will be necessary.

```pascal
type
  TInvoiceRec = record
    Ident: RawUTF8;
    Amount: currency;
  end;
TInvoiceRecs = array of TInvoiceRec;
TSQLInvoice = class(TSQLRecord)
protected
  fDetails: TInvoiceRecs;
  fTotal: Currency;
procedure SetDetails(const Value: TInvoiceRecs);
published
  property Details: TInvoiceRecs read fDetails write SetDetails;
  property Total: Currency read fTotal;
end;
```

Note that the Total property does not have any setter (aka write statement). So it will be read-only,
from the ORM point of view. In fact, the following protected method will compute the Total property content from the Details property values, when they will be modified:

```pascal
procedure TSQLInvoice.SetDetails(const Value: TInvoiceRecs);
var
  i: integer;
begin
  fDetails := Value;
  fTotal := 0;
  for i := 0 to high(Value) do
    fTotal := fTotal + Value[i].Amount;
end;
```

When the object content will be sent to the Server, the Total value of the JSON content sent will contain the expected value.

Note that with this implementation, the SetDetails must be called explicitly. That is, you should not only modify directly the Details[] array content, but either use a temporary array during edition then assign its value to Invoice.Details, or force the update with a line of code like:

```pascal
Invoice.Details := Invoice.Details; // force Total calculation
```

### 5.8.2. TSQLRecord.ComputeFieldsBeforeWrite

Even if a TSQLRecord instance should not normally have access to the TSQLRest level, according to OOP principles, the following virtual method have been defined:

```pascal
TSQLRecord = class(TObject)
public
  procedure ComputeFieldsBeforeWrite(aRest: TSQLRest; aOccasion: TSQLEvent); virtual;
(...)
```

It will be called automatically on the Client side, just before a TSQLRecord content will be sent to the remote server, before adding or update.

In fact, the TSQLRestClientURI.Add / Update / BatchAdd / BatchUpdate methods will call this method before calling TSQLRecord.GetJSONValues and send the JSON content to the server.

On the Server-side, in case of some business logic involving the ORM, the TSQLRestServer.Add / Update methods will also call ComputeFieldsBeforeWrite.

By default, this method will compute the TModTime / sftModTime and TCreateTime / sftCreateTime properties value from the current server time stamp, as such:

```pascal
procedure TSQLRecord.ComputeFieldsBeforeWrite(aRest: TSQLRest; aOccasion: TSQLEvent);
var
  F: integer;
begin
  if (self<>nil) and (aRest<>nil) then
    with RecordProps do begin
      if HasModTimeFields then
        for F := 0 to high(FieldType) do
          if FieldType[F]=sftModTime then
            SetInt64Prop(Self,Fields[F],aRest.ServerTimestamp);
      if HasCreateTimeField and (aOccasion=seAdd) then
        for F := 0 to high(FieldType) do
          if FieldType[F]=sftCreateTime then
            SetInt64Prop(Self,Fields[F],aRest.ServerTimestamp);
    end;
end;
```

You may override this method for you own purpose, saved the fact that you call this inherited implementation to properly handle TModTime and TCreateTime published properties.
5.9. Audit Trail for change tracking

Since most CRUD operations are centered within the scope of our mORMot server, we implemented in the ORM an integrated mean of tracking changes (aka Audit Trail) of any TSQLRecord.

Keeping a track of the history of business objects is one very common need for software modeling, and a must-have for any accurate data modeling, like Domain-Driven Design (page 99). By default, as expected by the OOP model, any change to an object will forget any previous state of this object. But thanks to mORMot’s exclusive change-tracking feature, you can persist the history of your objects.

5.9.1. Enabling audit-trail

By default, change-tracking feature will be disabled, saving performance and disk use. But you can enable change tracking for any class, by calling the following method, on server side:

```
aServer.TrackChanges([TSQLInvoice]);
```

This single line will let aServer: TSQLRestServer monitor all CRUD operations, and store all changes of the TSQLInvoice table within a TSQLRecordHistory table.

Since all content changes will be stored in this single table by default (note that the TrackChanges() method accepts an array of classes as parameters, and can be called several times), it may be handy to define several tables for history storage. Later on, an external database engine may be defined to store history, e.g. on cheap hardware (and big hard drives), whereas your main database may be powered by high-end hardware (and smaller SSDs) - see below (page 239). To do so, you define your custom class for history storage, then supply it as parameter:

```
type
  TSQLRecordSecondaryHistory = class(TSQLRecordHistory);
  (...)
aServer.TrackChanges([TSQLInvoice],TSQLRecordSecondaryHistory);
```

Then, all history will be stored in this TSQLRecordSecondaryHistory class (in its own table named SecondaryHistory), and not the default TSQLRecordHistory class (in its History table).

5.9.2. A true Time Machine for your objects

Once the object changes are tracked, you can later on browse the history of the object, by using the TSQLRecordHistory.CreateHistory(), then HistoryGetLast, HistoryCount, and HistoryGet() methods:

```
var
  aHist: TSQLRecordHistory;
  aInvoice: TSQLInvoice;
  aEvent: TSQLHistoryEvent; // will be either heAdd, heUpdate or heDelete
  aTimestamp: TModTime;
  (...
  aInvoice := TSQLInvoice.Create;
  aHist := TSQLRecordHistory.CreateHistory(aClient,TSQLInvoice,400);
  try
    writeln('Number of items in the record history: ',aHist.HistoryCount);
    for i := 0 to aHist.HistoryCount-1 do begin
      aHist.HistoryGet(i,aEvent,aTimestamp,aInvoice);
      writeln;
      writeln('Event: ',GetEnumName(TypeInfo(TSQLHistoryEvent),ord(aEvent)));
      writeln('Timestamp: ',TTimeLogBits(aTimestamp).ToText);
      writeln('Identifier: ',aInvoice.Number);
      writeln('Value: ',aInvoice.GetJSONValues(true,true,soSelect));
    end;
  finally
    aHist.Free;
    aInvoice.Free;
```
Note that you have several overloaded versions of TSQLRecordHistory.HistoryGet() to retrieve the record values.

As a result, our ORM is also transformed into a true *time machine*, for the objects which need it.

This feature will be available on both client and server sides, via the TSQLRecordHistory table.

### 5.9.3. Automatic history packing

This TSQLRecordHistory class will in fact create a History table in the main database, defined as such:

```
ID : TID
Event : TSQLHistoryEvent
History : TSQLRawBlob
ModifiedRecord : PtrInt
SentDataJSON : RawUTF8
Timestamp : TModTime
```

*History Record Layout*

In short, any modification via the ORM will be stored in the TSQLRecordHistory table, as a JSON object of the changed fields, in TSQLRecordHistory.SentDataJSON.

By design, direct SQL changes are not handled. If you run some SQL statements like DELETE FROM ... or UPDATE ... SET ... within your application or from any external program, then the History table won't be updated.

In fact, the ORM does not set any DB trigger to track low-level changes: it will slow down the process, and void the *persistence agnosticism* paradigm we want to follow, e.g. allowing to use a NoSQL database like MongoDB.

When the history grows, the JSON content may become huge, and fill the disk space with a lot of duplicated content. In order to save disk space, when a record reaches a define number of JSON data rows, all this JSON content is gathered and compressed into a BLOB, in TSQLRecordHistory.History.

You can force this packing process by calling TSQLRestServer.TrackChangesFlush() manually in your code. Calling this method will also have a welcome side effect: it will read the actual content of the record from the database, then add a fake helpUpdate event in the history if the field values do not match the one computed from tracked changes, to ensure that the audit trail will be correct. As a consequence, history will become always synchronized with the actual data persisted in the database, even if external SQL did by-pass the CRUD methods of the ORM, via unsafe DELETE FROM ... or UPDATE ... SET ... statements.

You can tune how packing is defined for a given TSQLRecord table, by using some optional parameters to the registering method:

```pascal
procedure TrackChanges(const aTable: array of TSQLRecordClass;
    aTableHistory: TSQLRecordHistoryClass=nil; aMaxHistoryRowBeforeBlob: integer=1000;
    aMaxHistoryRowPerRecord: integer=10; aMaxUncompressedBlobSize: integer=64*1024); virtual;
```

Take a look at the documentation of this method (or the comments in its declaration code) for further information.

Default options will let TSQLRestServer.TrackChangesFlush() be called after 1000 individual TSQLRecordHistory.SentDataJSON rows are stored, then will compress them into a BLOB once 10 JSON rows are available for a given record, ensuring that the uncompressed BLOB size for a single record won't use more than 64 KB of memory (but probably much less in the database, since it is
stored with very high compression rate).

### 5.10. Master/slave replication

As stated during *TSQlRecord fields definition* (page 131), the ORM is able to maintain a revision number for any TSQLRecord table, so that it the table may be easily synchronized remotely by another TSQLRestServer instance.

If you define a TRecordVersion published property, the ORM core will fill this field just before any write with a monotonically increasing revision number, and will take care of any deletion, so that those modifications may be replayed later on any other database.

This synchronization will work as a strict master/slave replication scheme, as a one-way on demand refresh of a replicated table. Each write operation on the master database on a given table may be easily reflected on one or several slave databases, with almost no speed nor storage size penalty.

In addition to this *on demand* synchronization, a real-time notification mechanism, using *WebSockets* communication - see below (page 444) - may be defined.

#### 5.10.1. Enable synchronization

In order to enable this replication mechanism, you should define a TRecordVersion published property in the TSQLRecord class type definition:

```pascal
TSQLRecordPeopleVersioned = class(TSQLRecordPeople)
protected
  fFirstName: RawUTF8;
  fLastName: RawUTF8;
  fVersion: TRecordVersion;
published
  property FirstName: RawUTF8 read fFirstName write fFirstName;
  property LastName: RawUTF8 read fLastName write fLastName;
  property Version: TRecordVersion read fVersion write fVersion;
end;
```

Only a single TRecordVersion field is allowed per TSQLRecord class - it will not mean anything to manage more than one field of this type.

Note that this field will be somewhat "hidden" to most ORM process: a regular TSQLRest.Retrieve won't fill this Version property, since it is an internal implementation detail. If you want to lookup its value, you will have to explicitly state its field name at retrieval. Any TRecordVersion is indeed considered as a "non simple field", just like BLOB fields, so will need explicit retrieval of its value.

In practice, any TSQLRest.Add and TSQLRest.Update on this TSQLRecordPeopleVersioned class will increase this Version revision number field, and a TSQLRest.Delete will populate an external TSQLRecordTableDelete table with the ID of the deleted record, associated with a TRecordVersion revision.

As consequences:

- The monotonistic TRecordVersion number is shared at TSQLRestServer level, among all tables containing a TRecordVersion published field;
- The TSQLRecordTableDelete table should be part of the TSQLModel, in conjunction with TSQLRecordPeopleVersioned;
- If the TSQLRecordTableDelete table is not part of the TSQLModel, the TSQLRestServer will add it - but you should better make it explicitly appearing in the data model;
- A single TSQLRecordTableDelete table will maintain the list of all deleted data rows, of all tables containing a TRecordVersion published field;
The TSQLRecordPeopleVersioned table appearance order in the TSQLModel will matter, since TSQLRecordTableDelete.ID will use this table index order in the database model to identify the table type of the deleted row - in a similar way to TRecordReference and TRecordReferenceToBeDeleted (page 139).

All the synchronization preparation will be taken care by the ORM kernel on its own, during any write operation. There is nothing particular to maintain or setup, in addition to this TRecordVersion field definition, and the global TSQLRecordTableDelete table.

5.10.2. From master to slave

To replicate this TSQLRecordPeopleVersioned table from another TSQLRestServer instance, just call the following method:

```
TSQLRecordPeopleVersioned, aClient);
```

This single line will request a remote server via a Client: TSQLRestClientURI connection (which may be over HTTP) for any pending modifications since its last call, then will fill the local aServer: TSQLRestServer database so that the local TSQLRecordPeopleVersioned table will contain the very same content as the remote master TSQLRestServer.

You can safely call TSQLRestServer.RecordVersionSynchronizeSlave from several clients, to replicate the master data in several databases.

Using a TTimer may increase responsiveness of a client application, and allow refresh of displayed data, with limited resources (e.g. with a 500 ms period, on a given screen).

Only the modified data will be transmitted over the wire, as two REST/JSON queries (one for the insertions/updates, another for the deletions), and all the local write process will use optimized BATCH writing - see below (page 350). This means that the synchronization process will try to use as minimal bandwidth and resources as possible, on both sides.

In practice, you may define the Master side as such:

```
MasterServer := TSQLRestServerDB.Create(MasterModel, 'master.db3');
HttpMasterServer := TSQLHttpServer.Create('8888', [MasterServer], '+', useBidirSocket);
```

On the Slave side, the HTTP client will access the Master database as usual:

```
MasterClient := TSQLHttpClientWebsockets.Create('127.0.0.1', HTTP_DEFAULTPORT, MasterModel);
```

Of course, the model should match for both MasterServer and MasterClient instances. This is why we used the same MasterModel variable name (probably defined in a shared unit).

Assuming that the slave database has been defined as such:

```
SlaveServer := TSQLRestServerDB.Create(SlaveModel, 'slave.db3');
```

Then you can run replication from the Slave side with a single line, for a given table:

```
SlaveServer.RecordVersionSynchronizeSlave(TSQLRecordPeopleVersioned, MasterClient);
```

This command will process the replication as such:
ORM Replication Classes via REST

Of course, the slaves should be considered as read-only, otherwise the version numbers may conflict, and the whole synchronization may become a failure.

But you can safely replicate servers in cascade, if needed: the version numbers will be propagated from masters to slaves, and the data will always be in a consistent way.
ORM Cascaded Replication Classes via REST

This cascading Master/Slave replication design may be used in conjunction with the CQRS pattern (Command Query Responsibility Segregation). In fact, the Slave 2 database may be a local read-only database instance, used only for reporting purposes, e.g. by marketing or management people, whereas the Slave 1 may be the active read-only database, on which all local business process will read their data. As such, the Slave 2 instance may be replicated much less often than than Slave 1 database - which may be even be replicated in real time, as we will now see.

5.10.3. Real-time synchronization

Sometimes, the on-demand synchronization is not enough.

For instance, you may need to:
- Synchronize a short list of always evolving items which should be reflected as soon as possible;
- Involve some kind of ACID-like behavior (e.g. handle money!) in your replicated data;
- Replicate not from a GUI application, but from a service, so use of a TTimer is not an option;
- Combine REST requests (for ORM or services) and master/slave ORM replication on the same wire, e.g. in a multi-threaded application;
- Use an *Event Oriented Persistence*, and expect to be notified from any change of state - see below (page 622).

In this case, the framework is able to use *WebSockets* and asynchronous callbacks to let the master/slave replication - see below (page 444) - take place without the need to ask explicitly for pending data. You will need to use TSQLRestServer.RecordVersionSynchronizeMasterStart, TSQLRestServer.RecordVersionSynchronizeSlaveStart and TSQLRestServer.RecordVersionSynchronizeSlaveStop methods over the proper kind of bidirectional connection.

The first requirement is to allow *WebSockets* on your *Master* HTTP server, so initialize the TSQLHttpServer class as a useBidirSocket kind of server - see below (page 325):

```
MasterServer := TSQLRestServerDB.Create(MasterModel,'master.db3');
HttpMasterServer := TSQLHttpServer.Create('8888',[MasterServer],'+',useBidirSocket);
HttpMasterServer.WebSocketsEnable(Server,'PrivateAESEncryptionKey');
```

On the *Slave* side, the HTTP client should also be upgraded to support *WebSockets*:

```
MasterClient := TSQLHttpClientWebsockets.Create('127.0.0.1',HTTP_DEFAULTPORT,MasterModel);
MasterClient.WebSocketsUpgrade('PrivateAESEncryptionKey');
```

Of course, the model should match for both MasterServer and MasterClient instances. As the *WebSockets* protocol definition - here above the same 'PrivateAESEncryptionKey' private key.

Then you enable the real-time replication service on the *Master* side:

```
MasterServer.RecordVersionSynchronizeMasterStart;
```

In practice, it will publish a IServiceRecordVersion interface-based service on the server side - see below (page 419).

Assuming that the *slave* database has been defined as such:

```
SlaveServer := TSQLRestServerDB.Create(SlaveModel,'slave.db3');
```

(in this case, the SlaveModel may not be the same as the MasterModel, but TSQLRecordPeopleVersioned should be part of both models)

Then you can initiate real-time replication from the *slave* side with a single line, for a given table:

```
SlaveServer.RecordVersionSynchronizeSlaveStart(TSQLRecordPeopleVersioned,MasterClient);
```

The above command will subscribe to the remote MasterSlave replication service (i.e. IServiceRecordVersion interface), to receive any change concerning the TSQLRecordPeopleVersioned ORM table, using the MasterClient connection via *WebSockets*, and persist all updates into the local SlaveServer database.

To stop the real-time notification for this ORM table, you could execute:

```
SlaveServer.RecordVersionSynchronizeSlaveStop(TSQLRecordPeopleVersioned);
```

Even if you do not call RecordVersionSynchronizeSlaveStop(), the replication will be stopped when the main SlaveServer instance will be released, and the MasterServer be *unsubscribe* this connection for its internal notification list.

This typical replication may be represented as such:
ORM Real-Time Replication Classes

The real-time notification details have been tuned, to consume as minimum bandwidth and resources as possible. For instance, if several modifications are to be notified on a slave connection in a short amount of time, the master is able to gather those modifications as a single WebSockets frame, which will be applied as a whole to the slave database, in a single BATCH transaction - see below (page 350).

5.10.4. Replication use cases

We may consider a very common corporate infrastructure:

Corporate Servers Replication

This kind of installation, with a main central office, and a network of local offices, will benefit from this master/slave replication. Simple redirection may be used - see below (page 233) - but it will expect the work to continue, even in case of Internet network failure. REST redirection will expect a 100%
connection up-link, which may be critical in some cases.

You could therefore implement replication in several ways:
- Either the main office is the master, and any write will be push to the Main Server, whereas local offices will have a replicated copy of the information - drawback is that in case of network failure, the local office will be limited to read only data access;

Corporate Servers Master/Slave Replication With All Data On Main Server
- Or each local office may host its own data in a dedicated table, synchronized as a master database; the main office will replicate (as a slave) the private data of each local server; in addition, all this data gathered by the Main Server may be further replicated to the other local offices, and be still accessible in read mode - in case of network failure, all the data is available on local servers, and the local private table is still writable.

Corporate Servers Master/Slave Replication With Private Local Data
Of course, the second solution seems preferable, even if a bit more difficult to implement. The ability of all local offices to work offline on their own private data, but still having all the other data accessible as read-only, will be a huge ROI.
As a benefit of using replication, the central main server will be less stressed, since most of the process will take place in local servers, and the main office server will only be used for shared data backup and read-only gathering of the other local databases. Only a small network bandwidth will be necessary (much less than a pure web solution), and CPU/storage resources will be minimal.

If needed, the *Real-time synchronization* (page 183) will allow to have the main office data replicated in "near real-time" in the local offices databases, whereas the write operations will still safely take place on the main Office. Another cascading replication may take place within any node, with a on-demand refresh, *e.g.* a 1 hour period, to implement the CQRS pattern (*Command Query Responsibility Segregation*).

Corporate Servers Master/Slave Replication With CQRS

Following the CQRS pattern, some demanding Queries may take place in those read-only "Reporting" replicated databases, without impacting the main local databases, in which all actual "Business" will take place.
6. Daily ORM

When you compare ORM and standard SQL, some aspects must be highlighted.

First, you do not have to worry about field orders and names, and can use field completion in the IDE. It is much more convenient to type Baby. then select the Name property, and access to its value.

The ORM code is much more readable than the SQL. You do not have to switch your mind from one syntax to another, in your code. Because SQL is a true language (see SQL Is A High-Level Scripting Language at http://www.fossil-scm.org/index.html/doc/tip/www/theory1.wiki). You can even forget about the SQL itself for most projects; only some performance-related or complex queries should be written in SQL, but you will avoid it most of the time. Think object pascal. And happy coding. Your software architecture will thank you for it.

Another good impact is the naming consistency. For example, what about if you want to rename your table? Just change the class definition, and your IDE will do all refactoring for you, without any risk of missing a hidden SQL statement anywhere. Do you want to rename or delete a field? Change the class definition, and the Delphi compiler will let you know all places where this property was used in your code. Do you want to add a field to an existing database? Just add the property definition, and the framework will create the missing field in the database schema for you.

Another risk-related improvement is about the strong type checking, included into the Delphi language during compile time, and only during execution time for the SQL. You will avoid most runtime exceptions for your database access: your clients will thank you for that. In one word, forget about field typing mismatch or wrong type assignment in your database tables. Strong typing is great in such cases for code SQA, and if you worked with some scripting languages (like JavaScript, Python or Ruby), you should have wished to have this feature in your project!
It is worth noting that our framework allows writing triggers and stored procedures (or like stored procedures) in Delphi code, and can create key indexing and perform foreign key checking in class definition.

Another interesting feature is the enhanced Grid component supplied with this framework, and the AJAX-ready orientation, by using natively JSON flows for Client-Server data streaming. The REST protocol can be used in most application, since the framework provides you with an easy to use "Refresh" and caching mechanism. You can even work off line, with a local database replication of the remote data.

For Client-Server - see below (page 295) - you do not have to open a connection to the database, just create an instance of a TSQLRestClient object (with the communication layer you want to use: direct access, Windows Messages, named pipe or HTTP), and use it as any normal Delphi object. All the SQL coding or communication and error handling will be done by the framework. The same code can be used in the Client or Server side: the parent TSQLRest object is available on both sides, and its properties and methods are strong enough to access the data.

6.1. ORM is not Database

It is worth emphasizing that you should not think about the ORM like a mapping of an existing DB schema. This is an usual mistake in ORM design.

The database is just one way of your objects persistence:
- Don't think about tables with simple types (text/number...), but objects with high level types;
- Don't think about Master/Detail, but logical units;
- Don't think "SQL", think about classes;
- Don't wonder "How will I store it?", but "Which data do I need?".

For instance, don't be tempted to always create a pivot table (via a TSQLRecordMany property), but consider using a dynamic array, TPersistent, TStrings or TCollection published properties instead.

Or consider that you can use a TRecordReference property pointing to any registered class of the TSQLModel, instead of creating one TSQLRecord property per potential table.

The mORMot framework is even able to persist the object without any SQL database, e.g. via TSQLRestStorageInMemory. In fact, its ORM core is optimized but not tied to SQL.

6.1.1. Objects, not tables

With an ORM, you should usually define fewer tables than in a "regular" relational database, because you can use the high-level type of the TSQLRecord properties to handle some per-row data.

The first point, which may be shocking for a database architect, is that you should better not create Master/Detail tables, but just one "master" object with the details stored within, as JSON, via dynamic array, TPersistent, TStrings or TCollection properties.

Another point is that a table is not to be created for every aspect of your software configuration. Let's confess that some DB architects design one configuration table per module or per data table. In an ORM, you could design a configuration class, then use the unique corresponding table to store all configuration encoded as some JSON data, or some DFM-like data. And do not hesitate to separate the configuration from the data, for all not data-related configuration - see e.g. how the mORMotOptions unit works. With our framework, you can serialize directly any TSQLRecord or...
TPersistent instance into JSON, without the need of adding this TSQLRecord to the TSQLModel list. Since revision 1.13 of the framework, you can even define TPersistent published properties in your TSQLRecord class, and it will be automatically serialized as TEXT in the database.

6.1.2. Methods, not SQL

At first, you should be tempted to write code as such (this code sample was posted on our forum, and is not bad code, just not using the ORM orientation of the framework):

```delphi
DrivesModel := CreateDrivesModel();
GlobalClient := TSQLRestClientDB.Create(DrivesModel, CreateDrivesModel(), 'drives.sqlite',
TSQLRestServerDB);
TSQLRestClientDB(GlobalClient).Server.DB.Execute('CREATE TABLE IF NOT EXISTS drives '(id INTEGER PRIMARY KEY, drive TEXT NOT NULL UNIQUE COLLATE NOCASE);');
for X := 'A' to 'Z' do
begin
TSQLRestClientDB(GlobalClient).Server.DB.Execute('INSERT OR IGNORE INTO drives (drive) VALUES ('' + StringToUTF8(X) + ':')
end;
```

Please, don't do that!

The correct ORM-oriented implementation should be the following:

```delphi
DrivesModel := TSQLModel.Create([TDrives], 'root');
GlobalClient := TMyClient.Create(DrivesModel, nil, 'drives.sqlite', TSQLRestServerDB);
GlobalClient.CreateMissingTables();
if GlobalClient.TableRowCount(TDrives)=0 then
begin
  D := TDrives.Create;
  try
    for X := 'A' to 'Z' do
    begin
      D.Drive := X;
      GlobalClient.Add(D, true);
    end;
  finally
    D.Free;
  end;
end;
```

In the above lines, no SQL was written. It is up to the ORM to:
- Create all missing tables, via the CreateMissingTables method - and not compute by hand a "CREATE TABLE IF NOT EXISTS..." SQL statement;
- Check if there is some rows of data, via the TableRowCount method - instead of a "SELECT COUNT(*) FROM DRIVES";
- Append some data using an high-level TDrives Delphi instance and the Add method - and not any "INSERT OR IGNORE INTO DRIVES".

Then, in order to retrieve some data, you'll be tempted to code something like that (extracted from the same forum article):

```delphi
procedure TMyClient.FillDrives(aList: TStrings);
var
  table: TSQLTableJSON;
  X, FieldIndex: Integer;
begin
  table := TSQLRestClientDB(GlobalClient).ExecuteList([TSQLDrives], 'SELECT * FROM drives');
  if (table <> nil) then
  try
    FieldIndex := table.FieldIndex('drive');
    if (FieldIndex >= 0) then
```
for X := 1 to table.RowCount do
    Items.Add(UTF8ToString(table.GetU(X, FieldIndex)));
finally
    table.Free;
end;
end;

Thanks to the TSQLTableJSON class, code is somewhat easy to follow. Using a temporary FieldIndex variable make also it fast inside the loop execution.

But it could also be coded as such, using the CreateAndFillPrepare then FillOne method in a loop:

procedure TMyClient.FillDrives(aList: TStrings);
begin
    aList.BeginUpdate;
    try
        aList.Clear;
        with TSQLDrives.CreateAndFillPrepare(GlobalClient,'') do
            try
                while FillOne do
                    aList.Add(UTF8ToString(Drive));
            finally
                Free;
            end;
    finally
        aList.EndUpdate;
    end;
end;

We even added the BeginUpdate / EndUpdate VCL methods, to have even cleaner and faster code (if you work with a TListBox e.g.).

Note that in the above code, an hidden TSQLTableJSON class is created in order to retrieve the data from the server. The abstraction introduced by the ORM methods makes the code not slowest, but less error-prone (e.g. Drive is now a RawUTF8 property), and easier to understand.

But ORM is not perfect in all cases.

For instance, if the Drive field is the only column content to retrieve, it could make sense to ask only for this very column. One drawback of the CreateAndFillPrepare method is that, by default, it retrieves all columns content from the server, even if you need only one. This is a common potential issue of an ORM: since the library doesn't know which data is needed, it will retrieve all object data, which is some cases is not worth it.

You can specify the optional aCustomFieldsCSV parameter as such, in order to retrieve only the Drive property content, and potentially save some bandwidth:

    with TSQLDrives.CreateAndFillPrepare(GlobalClient,'','Drive') do

Note that for this particular case, you have an even more high-level method, handling directly a TStrings property as the recipient:

procedure TMyClient.FillDrives(aList: TStrings);
begin
    GlobalClients.OneFieldValues(TSQLDrives,'drive','',aList);
end;

The whole query is made in one line, with no SELECT statement to write.

For a particular ID range, you may have written, with a specific WHERE clause using a prepared statement:

    GlobalClients.OneFieldValues(TSQLDrives,'drive',
        'ID>=? AND ID<=?',[],[aFirstID,aLastID],aList);
It is certainly worth reading all the (verbose) interface part of the mORMot.pas unit, e.g. the TSQLRest class, to make your own idea about all the high-level methods available. In the following pages, you'll find all needed documentation about this particular unit. Since our framework is used in real applications, most useful methods should already have been made available. If you need additional high-level features, feel free to ask for them, if possible with source code sample, in our forum, freely available at https://synopse.info..

6.1.3. Think multi-tier
And do not forget the framework is able to have several level of objects, thanks to our Client-Server architecture - see below (page 295). Such usage is not only possible, but strongly encouraged.

You should have business-logic level objects at the Client side. Then both business-logic and DB objects at the Server side.

If you have a very specific database schema, business-logic objects can be of very high level, encapsulating some SQL views for reading, and accessed via some RESTful service commands for writing - see below (page 372).

Another possibility to access your high-level type, is to use either custom SQLite3 SQL functions or stored procedures - see below (page 363) - both coded in Delphi.

6.2. One ORM to rule them all
Just before entering deeper into the mORMot material in the following pages (Database layer, Client-Server, Services), you may find out that this implementation may sounds restricted.

Some common (and founded) criticisms are the following (quoting from our forum):
- "One of the things I don't like so much about your approach to the ORM is the mis-use of existing Delphi constructs like "index n" attribute for the maximum length of a string-property. Other ORMs solve this i.e. with official Class-attributes";
- "You have to inherit from TSQLRecord, and can't persist any plain class";
- "There is no way to easily map an existing complex database".

Those concerns are pretty understandable. Our mORMot framework is not meant to fit any purpose, but it is worth understanding why it has been implemented as such, and why it may be quite unique within the family of ORMs - which almost all are following the Hibernate way of doing.

6.2.1. Rude class definition
Attributes do appear in Delphi 2010, and it is worth saying that FPC has an alternative syntax. Older versions of Delphi (still very deployed) do not have attributes available in the language, so it was not possible to be compatible with Delphi 6 up to latest versions (as we wished for our units).

It is perfectly right to speak about 'mis-use of index' - but this was the easiest and only way we found out to have such information, just using RTTI. Since this parameter was ignored and not used for most classes, it was re-used (also for dynamic array properties, to have faster lookup).

There is another "mis-use" for the "stored AS_UNIQUE" property, which is used to identify unique mandatory columns.

Using attributes is one of the most common ways of describing tables in most ORMs. On the other hand, some coders have a concern about such class definitions. They are mixing DB and logic: you are somewhat polluting the business-level class definition with DB-related stuff.
That is why other kind of ORMs provide a way of mapping classes to tables using external files (some ORMs provide both ways of definition). And why those days, even code gurus identified the attributes overuse as a potential weakness of code maintainability. Attributes do have a huge downsize, when you are dealing with a Client-Server ORM, like ours: on the Client side, those attributes are pointless (client does not need to know anything about the database), and you need to link to all the DB plumbing code to your application. For mORMot, it was some kind of strong argument.

For the very same reasons, the column definitions (uniqueness, indexes, required) are managed in mORMot at two levels:
- At ORM level for DB related stuff (like indexes, which is a DB feature, not a business feature);
- At Model level for Business related stuff (like uniqueness, validators and filters).

When you take a look at the supplied validators and filters - see Filtering and Validating (page 172) - you'll find out that this is much powerful than the attributes available in "classic" ORMs: how could you validate an entry to be an email, or to match a pattern, or to ensure that it will be stored in uppercase within the DB?

Other question worth asking is about the security. If you access the data remotely, a global access to the DB is certainly not enough. Our framework handle per-table CRUD level access for its ORM, above the DB layer (and has also complete security attributes for services) - see below (page 544). It works however the underneath DB grants are defined (even an DB with no user rights - like in-memory or SQLite3 is able to do it).

The mORMot point of view (which is not the only one), is to let the DB persist the data, as safe and efficient as possible, but rely on higher levels layers to implement the business logic. The framework favors convention over configuration, which is known to save a lot of time (if you use WCF on a daily basis, as I do, you and your support team know about the .config syndrome). It will make it pretty database-agnostic (you can even not use a SQL database at all), and will make the framework code easier to debug and maintain, since we don't have to deal with all the DB engine particularities. In short, this is the REST point of view, and main cause of success: CRUD is enough in any KISS-friendly design.

6.2.2. Persist TSQLRecord, not any class

About the fact that you need to inherit from TSQLRecord, and can’t persist any PODO (Plain Old Delphi Object), our purpose was in fact very similar to the "Layer Supertype" pattern of Domain-Driven-Design, as explained by Martin Fowler: It is not uncommon for all the objects in a layer to have methods you don’t want to have duplicated throughout the system. You can move all of this behavior into a common Layer Supertype.

In fact, for TSQLRecord / TSQLRest / ORM remote access, you have already all Client-Server CRUD operations available. Those classes are abstract common Supertypes, ready to be used in your projects. It has been optimized a lot (e.g. with a cache and other nice features), so I do not think reinventing a CRUD / database service is worth the prize. You have secure access to the ORM classes, with user/group attributes. Almost everything is created by code, just from the TSQLRecord class definition, via RTTI. So it may be faster (and safer) to rely on it, than defining all your class hierarchy by hand.

To be fair, most DDD frameworks for Java or C# expect e.g. Entities classes to inherit from a given Entity class, or add class attributes to the POJO/POCO to define the persistence details. So we are not the single one in this case!
But the concern of not being able to persist any class (it needs to inherit from TSQLRecord) does perfectly make sense. Especially in the context of DDD modeling, where the DDD objects will benefit from being uncoupled from the framework, which may pollute the domain logic. All those expectations tend to break the Persistence Ignorance principle, as requested by DDD patterns.

This is why we added to the framework the ability to persist any plain class, using Repository services, but still using the ORM under the hood, for the actual persistence on any SQL or NoSQL database engine. The TSQLRecord can be generated from any Delphi persistent class, then an automated mapping is maintained by mORMot between both class instances. Data access is then defined as clean CQRS Repository Services.

For instance, a TUser class may be persisted via such a service:

```delphi
type
  IDomUserCommand = interface(IDomUserQuery)
  ['{D345854F-7337-4006-B324-SD635F8ED312}']
  function Add(const aAggregate: TUser): TCQRSResult;
  function Update(const aUpdatedAggregate: TUser): TCQRSResult;
  function Delete: TCQRSResult;
  function Commit: TCQRSResult;
end;
```

Here, the write operations are defined in an IDomUserCommand service, which is separated (but inherits) from IDomUserQuery, which is used for read operations. See below (page 610) for more details about this feature.

### 6.2.3. Several ORMs at once

To be clear, mORMot offers several kind of table definitions:

- Via TSQLRecord / TSQLRecordVirtual "native ORM" classes: data storage is using either fast in-memory lists via TSQLRestStorageInMemory, or SQLite3 tables (in memory, on file, or virtual). In this case, we do not use index for strings (column length is not used by any of those engines).

- Via TSQLRecord "external ORM-managed" classes: after registration via a call to the VirtualTableExternalRegister() / VirtualTableExternalMap() functions, external DB tables are created and managed by the ORM, via SQL - see below (page 239). These classes will allow creation of tables in any supported database engine - currently SQLite3, Oracle, Jet/MSAccess, MS SQL, Firebird, DB2, PostgreSQL, MySQL, Informix and NexusDB - via whatever OleDB, ODBC / ZDBC provider, or any DB.pas unit. For the "external ORM-managed" TSQLRecord type definitions, the ORM expects to find an index attribute for any text column length (i.e. RawUTF8 or string published properties). This is the only needed parameter to be defined for such a basic implementation, in regard to TSQLRecord kind of classes. Then can specify addition field/column mapping, if needed.

- Via TSQLRecord "external ODM-managed" classes: after registration via a call to the StaticMongoDBRegister() or StaticMongoDBRegisterAll() functions, external MongoDB collections are created and managed via NoSQL and Object-Document Mapping (ODM) (page 96). In this case, no index attribute for setting text column length is necessary.

- Via TSQLRecordMappedAutoID / TSQLRecordMappedForcedID "external mapped" classes: DB tables are not created by the ORM, but already existing in the DB, with sometimes a very complex layout. This feature is not yet implemented, but on the road-map. For this kind of classes we won't probably use attributes, nor even external files, but we will rely on definition from code, either with a fluent definition, or with dedicated classes (or interface).

- Via any kind of Delphi class, mapped to their internal TSQLRecord class, using CQRS Repository Services as presented below (page 610).

Why have several database back-end at the same time?
Most of the existing software architecture rely on one dedicated database per domain, since it is more convenient to administrate one single server. But there are some cases when it does make sense to have several databases at once.

In practice, when your data starts to grow, you may need to archive older data in a dedicated remote database, e.g. using cheap storage (bunch of Hard Drives in RAID). Since this data will be seldom retrieved, it is not an issue to have slower access time. And you will be able to keep your most recent data accessible in a local high-speed engine (running on SSD).

Another pattern is to use dedicated consolidation DBs for any analysis. In fact, SQL normalization is good for most common relation work, but sometimes denormalization is necessary, e.g. for statistic or business analyse purposes. In this case, dedicated consolidation databases, containing the data already prepared and indexed in a ready-to-use denormalized layout.

Last but not least, some Event Sourcing architectures even expect several DB back-end at once:
- It will store the status on one database (e.g. high-performance in-memory) for most common requests to be immediate;
- And store the modification events in another ACID database (e.g. SQLite3, Oracle, Jet/MSAccess, MS SQL, Firebird, DB2, PostgreSQL, MySQL, Informix or NexusDB), even a high-speed NoSQL engine like MongoDB.

It is possible to directly access ORM objects remotely (e.g. the consolidation DB), mostly in a read-only way, for dedicated reporting, e.g. from consolidated data - this is one potential CQRS implementation pattern with mORMot. Thanks to the framework security, remote access will be safe: your clients won't be able to change the consolidation DB content!

As can be easily guessed, such design models are far away from a basic ORM built only for class persistence. And mORMot’s ORM/ODM offers you all those possibilities.

### 6.2.4. The best ORM is the one you need

Therefore, we may sum up some potential use of ORM, depending of your intent:
- If you want to persist some data objects (not tied to complex business logic), the framework's ORM will be a light and fast candidate, targetting SQLite3, Oracle, Jet/MSAccess, MS SQL, Firebird, DB2, PostgreSQL, MySQL, Informix, NexusDB databases, or even with no SQL engine, using TSQLRestStorageInMemory class which is able to persist its content in small files - see below (page 230);
- If your understanding of ORM is just to persist some existing objects with associated business code, mORMot could help you, thanks to its Repository Services automatically generated over TSQLRecord, as presented below (page 610);
- If you want a very fast low-level Client-Server layer, mORMot is a first class candidate: we identified that some users are using the built-in JSON serialization and HTTP server features to create their application, using a RESTful/ SOA architecture - see below (page 373) and below (page 419);
- If your expectation is to map an existing complex RDBMS, mORMot will allow to publish existing SQL statements as services, using e.g. Interface-based services - see below (page 419) - over optimized SynDB.pas data access - see below (page 241) - as explained in Legacy code and existing projects (page 77);
- If you need (perhaps not now, but probably in the future) to create some kind of scalable Domain-Driven design architecture, you'll have all needed features at hand with mORMot;

Therefore, mORMot is not just an ORM, nor just a "classic" ORM.
7. Database layer

7.1. SQLite3-powered, not SQLite3-limited

The core database of this framework uses the SQLite3 library, which is a Free, Secure, Zero-Configuration, Server-less, Single Stable Cross-Platform Database File database engine.

As stated below, you can use any other database access layer, if you wish:
- A fast in-memory engine is included, which outperforms any SQL-based solution in terms of speed - but to the price of a non ACID behavior on disk (but ACID in RAM);
- An integrated SQLite3 engine, which is the best candidate for an embedded solution, even on server side;
- Any remote RDBMS database, via one or more OleDB, ODBC, Zeos or Oracle connections to store your precious ORM objects. Or you can use any DB.pas unit, e.g. to access NexusDB or any database engines supported by DBExpress, FireDAC, AnyDAC, UniDAC (or the deprecated BDE). In all cases, the ORM supports currently SQLite3, Oracle, Jet/MSAccess, MS SQL, Firebird, DB2, PostgreSQL, MySQL, Informix and NexusDB SQL dialects;
- Any other TSQLRest instance (either another TSQLRestServer, or a remote TSQLRestClientHTTP) - see below (page 233);
- Direct access to a MongoDB database, which implements a true NoSQL and Object-Document Mapping (ODM) (page 96) design.
SQLite3 will be used as the main SQL engine, able to JOIN all those tables, thanks to its Virtual Table unique feature. You can in fact mix internal and external engines, in the same database model, and access all data in one unique SQL statement.

### 7.1.1. SQLite3 as core

This framework uses a compiled version of the official SQLite3 library source code, and includes it natively into Delphi code. This framework therefore adds some very useful capabilities to the Standard SQLite3 database engine, but keeping all its advantages, as listed in the previous paragraph of this document:

- Can be either statically linked to the executable, or load external sqlite3.dll;
- Faster database access, through unified memory model, and usage of the FastMM4 memory manager (which is almost 10 times faster than the default Windows memory manager for memory allocation);
- Optional direct encryption of the data on the disk (up to AES256 level, that is Top-Secret security);
- Use via mORMot’s ORM let database layout be declared once in the Delphi source code (as published properties of classes), avoiding most SQL writing, hence common field or table names mismatch;
- Locking of the database at the record level (SQLite3 only handles file-level locking);
- Of course, the main enhancement added to the SQLite3 engine is that it can be deployed in a stand-alone or Client-Server architecture, whereas the default SQLite3 library works only in stand-alone mode.

From the technical point of view, here are the current compilation options used for building the SQLite3 engine:

- Uses ISO 8601:2004 format to properly handle date/time values in TEXT field, or in faster and smaller Int64 custom types: TTimeLog / TModTime / TCreateTime or TUnixTime / TUnixMSTime;
- SQLite3 library unit was compiled including RTREE extension for doing very fast range queries;
- It can include FTS3/FTS4 full text search engine (MATCH operator), with integrated SQL optimized ranking function;
- The framework makes use only of newest API (sqlite3_prepare_v2) and follows latest SQLite3 official documentation;
- Additional collations (i.e. sorting functions) were added to handle efficiently not only UTF-8 text, but also e.g. ISO 8601 time encoding, fast Win1252 diacritic-agnostic comparison and native slower but accurate Windows UTF-16 functions;
- Additional SQL functions like Soundex for English/French/Spanish phonemes, MOD or CONCAT, and some dedicated functions able to directly search for data within BLOB fields containing an Delphi high-level type (like a serialized dynamic array);
- Additional REGEXP operator/function using the Open Source PCRE library to perform regular expression queries in SQL statements;
- Custom SQL functions can be defined in Delphi code;
- Automatic SQL statement parameter preparation, for execution speed up;
- TSQLDatabase can cache the last results for SELECT statements, or use a tuned client-side or server-side per-record caching, in order to speed up most read queries, for lighter web server or client User Interface e.g.;
- User authentication handling (SQLite3 is user-free designed);
- SQLite3 source code was compiled without thread mutex: the caller has to be thread-safe aware; this is faster on most configurations, since mutex has to be acquired once): low level sqlite3_*() functions are not thread-safe, as TSQLRequest and TSQLBlobStream which just wrap them; but TSQLDataBase is thread-safe, as TSQLTableDB/TSQLRestServerDB/TSQLRestClientDB which call TSQLDataBase;
- Compiled with SQLITE_OMIT_SHARED_CACHE define, since with the new Client-Server approach of this framework, no concurrent access could happen, and an internal efficient caching algorithm is added, avoiding most call of the SQLite3 engine in multi-user environment (any AJAX usage should benefit of it);
- The embedded SQLite3 database engine can be easily updated from the official SQLite3 source code available at https://sqlite.org.. - use the amalgamation C file with a few minor changes (documented in the SynSQLite3Static.pas unit) - the resulting C source code delivered as .obj/.o is also available in the official Synopse source code repository.

The overhead of including SQLite3 in your server application will be worth it: just around 1 MB to the executable, but with so many nice features, even if only external databases are used.

7.1.2. Extended by SQLite3 virtual tables

Since the framework is truly object oriented, another database engine could be used instead of the framework. You could easily write your own TSQLRestServer descendant (as an example, we included a fast in-memory database engine as TSQLRestServerFullMemory) and link to a another engine (like FireBird, or a private one). You can even use our framework without any link to the SQLite3 engine itself, via our provided very fast in memory dataset (which can be made persistent by writing and reading JSON files on disk). The SQLite3 engine is implemented in a separate unit, named SynSQLite3.pas, and the main unit of the framework is mORMot.pas. A bridge between the two units is made with mORMotSQLite3.pas, which will found our ORM framework using SQLite3 as its core.

The framework ORM is able to access any database class (internal or external), via the powerful SQLite3 Virtual Table mechanisms - see below (page 225). For instance, any external database (via OleDB / ODBC / ZDBC providers or direct Oracle connection) can be accessed via our SynDB.pas-based dedicated units, as stated below (page 239).

As a result, the framework has several potential database back-ends, in addition to the default SQLite3 file-based engine. Each engine may have its own purpose, according to the application expectations.
Currently SQLite3, Oracle, Jet/MSAccess, MS SQL, Firebird, DB2, PostgreSQL, MySQL, Informix and NexusDB SQL dialects are handled by our ORM.

7.1.3. Data access benchmark

Purpose here is not to say that one library or database is better or faster than another, but publish a snapshot of mORMot persistence layer abilities, depending on each access library.

In this timing, we do not benchmark only the "pure" SQL/DB layer access (SynDB.pas units), but the whole Client-Server ORM of our framework.

Process below includes all aspects of our ORM:

- Access via high level CRUD methods (Add/Update/Delete/Retrieve, either per-object or in BATCH mode);
- Read and write access of TSQLRecord instances, via optimized RTTI;
- JSON marshalling of all values (ready to be transmitted over a network);
- REST routing, with security, logging and statistic;
- Virtual cross-database layer using its SQLite3 kernel;
- SQL on-the-fly generation and translation (in virtual mode);
- Access to the database engines via several libraries or providers.

In those tests, we just bypassed the communication layer, since TSQLRestClient and TSQLRestServer are run in-process, in the same thread - a TSQLRestServerDB instance. So you have here some raw performance testimony of our framework's ORM and RESTful core, and may expect good scaling abilities when running on high-end hardware, over a network.

On a recent notebook computer (Core i7 and SSD drive), depending on the back-end database interfaced, mORMot excels in speed, as will show the following benchmark:

- You can persist up to 570,000 objects per second, or retrieve 870,000 objects per second (for our pure Delphi in-memory engine);
- When data is retrieved from server or client ORM Cache (page 174), you can read more than 900,000 objects per second, whatever the database back-end is;
- With a high-performance database like Oracle, and our direct access classes, you can write 70,000 (via array binding) and read 160,000 objects per second, over a 100 MB network;
- When using alternate database access libraries (e.g. Zeos, or DB.pas based classes), speed is lower (even if comparable for DB2, MS SQL, PostgreSQL, MySQL) but still enough for most work, due to some optimizations in the mORMot code (e.g. caching of prepared statements, SQL multi-values insertion, direct export to/from JSON, SQLite3 virtual mode design, avoid most temporary memory allocation...).

Difficult to find a faster ORM, I suspect.

7.1.3.1. Software and hardware configuration

The following tables try to sum up all available possibilities, and give some benchmark (average objects/second for writing or reading).

In these tables:
- 'SQLite3 (file full/off/exc)' indicates use of the internal SQLite3 engine, with or without Synchronous := smOff and/or DB::LockingMode := lmExclusive - see below (page 222);
- 'SQLite3 (mem)' stands for the internal SQLite3 engine running in memory;
- 'SQLite3 (ext ...)' is about access to a SQLite3 engine as external database - see below (page 239),

```markdown
<table>
<thead>
<tr>
<th>Database Type</th>
<th>Writing Speed</th>
<th>Reading Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite3 (file)</td>
<td>570,000</td>
<td>870,000</td>
</tr>
<tr>
<td>SQLite3 (mem)</td>
<td>900,000</td>
<td>160,000</td>
</tr>
<tr>
<td>SQLite3 (ext)</td>
<td>70,000 (array)</td>
<td>160,000 (network)</td>
</tr>
</tbody>
</table>
```
either as file or memory;
- ' TObjectList' indicates a TSQLRestStorageInMemory instance - see below (page 230) - either static (with no SQL support) or virtual (i.e. SQL featured via SQLite3 virtual table mechanism) which may persist the data on disk as JSON or compressed binary;
- 'WinHTTP SQLite3' and 'Sockets SQLite3' stands for a SQLite3 engine published over HTTP using our SynDBRemote.pas unit using the winHTTP API or plain sockets on the client side - see below (page 263), then accessed as an external database by our ORM;
- 'NexusDB' is the free embedded edition, available from official site;
- 'Jet' stands for a Jet/MSAccess database engine, accessed via OleDB.
- 'Oracle' shows the results of our direct OCI access layer (SynDBOracle.pas);
- 'Zeos *' indicates that the database was accessed directly via the ZDBC layer;
- 'FireDAC *' stands for FireDAC library;
- 'UniDAC *' stands for UniDAC library;
- 'BDE *' when using a BDE connection;
- 'ODBC *' for a direct access to ODBC;
- 'MongoDB ack/no ack' for direct MongoDB access (SynMongoDB.pas) with or without write acknowledge.

This list of database providers is to be extended in the future. Any feedback is welcome!

Numbers are expressed in rows/second (or objects/second). This benchmark was compiled with Delphi XE4, since newer compilers tends to give better results, mainly thanks to function in-lining (which was not existing e.g. in Delphi 6-7).

Note that these tests are not about the relative speed of each database engine, but reflect the current status of the integration of several DB libraries within the mORMot database access.

Benchmark was run on a Core i7 notebook, running Windows 7, with a standard SSD, including anti-virus and background applications:
- Linked to a shared Oracle 11.2.0.1 database over 100 Mb Ethernet;
- MS SQL Express 2008 R2 running locally in 64-bit mode;
- IBM DB2 Express-C edition 10.5 running locally in 64-bit mode;
- PostgreSQL 9.2.7 running locally in 64-bit mode;
- MySQL 5.6.16 running locally in 64-bit mode;
- Firebird embedded in revision 2.5.2;
- NexusDB 3.11 in Free Embedded Version;
- MongoDB 2.6 in 64-bit mode.

So it was a development environment, very similar to low-cost production site, not dedicated to give best performance. During the process, CPU was noticeable used only for SQLite3 in-memory and TObjectList - most of the time, the bottleneck is not the CPU, but the storage or network. As a result, rates and timing may vary depending on network and server load, but you get results similar to what could be expected on customer side, with an average hardware configuration. When using high-head servers and storage, running on a tuned Linux configuration, you can expect even better numbers.

Tests were compiled with the Delphi XE4 32-bit mode target platform. Most of the tests do pass when compiled as a 64-bit executable, with the exception of some providers (like Jet), not available on this platform. Speed results are almost the same, only slightly slower; so we won't show them here.

You can compile the "15 - External DB performance" supplied sample code, and run the very same benchmark on your own configuration. Feedback is welcome!

From our tests, the UniDAC version we were using had huge stability issues when used with DB2: the
tests did not pass, and the DB2 server just hang processing the queries, whereas there was no problem with other libraries. It may have been fixed since, but you won't find any "UniDAC DB2" results in the benchmark below in the meanwhile.

7.1.3.2. Insertion speed

Here we insert 5,000 rows of data, with diverse scenarios:
- 'Direct' stands for a individual `Client.Add()` insertion;
- 'Batch' mode will be described below (page 350);
- 'Trans' indicates that all insertion is nested within a transaction - which makes a great difference, e.g. with a SQLite3 database.

Here are some insertion speed values, in objects/second:

<table>
<thead>
<tr>
<th></th>
<th>Direct</th>
<th>Batch</th>
<th>Trans</th>
<th>Batch Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite3 (file full)</td>
<td>462</td>
<td>28123</td>
<td>84823</td>
<td>181455</td>
</tr>
<tr>
<td>SQLite3 (file off)</td>
<td>2102</td>
<td>83093</td>
<td>88006</td>
<td>202667</td>
</tr>
<tr>
<td>SQLite3 (file off exc)</td>
<td>28847</td>
<td>193453</td>
<td>89451</td>
<td>207615</td>
</tr>
<tr>
<td>SQLite3 (mem)</td>
<td>89456</td>
<td>236540</td>
<td>104249</td>
<td>239165</td>
</tr>
<tr>
<td>TObjectList (static)</td>
<td>314465</td>
<td>543892</td>
<td>298846</td>
<td>542652</td>
</tr>
<tr>
<td>TObjectList (virtual)</td>
<td>325393</td>
<td>545672</td>
<td>326370</td>
<td>545018</td>
</tr>
<tr>
<td>SQLite3 (ext full)</td>
<td>424</td>
<td>14523</td>
<td>102049</td>
<td>164636</td>
</tr>
<tr>
<td>SQLite3 (ext off)</td>
<td>2245</td>
<td>47961</td>
<td>109706</td>
<td>189250</td>
</tr>
<tr>
<td>SQLite3 (ext off exc)</td>
<td>41589</td>
<td>180759</td>
<td>108481</td>
<td>192071</td>
</tr>
<tr>
<td>SQLite3 (ext mem)</td>
<td>101440</td>
<td>211389</td>
<td>113530</td>
<td>209713</td>
</tr>
<tr>
<td>WinHTTP SQLite3</td>
<td>2165</td>
<td>36464</td>
<td>2079</td>
<td>38478</td>
</tr>
<tr>
<td>Sockets SQLite3</td>
<td>8118</td>
<td>75251</td>
<td>8553</td>
<td>80550</td>
</tr>
<tr>
<td>MongoDB (ack)</td>
<td>10081</td>
<td>84585</td>
<td>9800</td>
<td>85232</td>
</tr>
<tr>
<td>MongoDB (no ack)</td>
<td>33223</td>
<td>273672</td>
<td>34665</td>
<td>274393</td>
</tr>
<tr>
<td>ODBC SQLite3</td>
<td>492</td>
<td>11746</td>
<td>35367</td>
<td>82425</td>
</tr>
<tr>
<td>ZEOS SQLite3</td>
<td>494</td>
<td>11851</td>
<td>56206</td>
<td>85705</td>
</tr>
<tr>
<td>FireDAC SQLite3</td>
<td>20605</td>
<td>38853</td>
<td>40042</td>
<td>113752</td>
</tr>
<tr>
<td>UniDAC SQLite3</td>
<td>477</td>
<td>8725</td>
<td>26552</td>
<td>38756</td>
</tr>
<tr>
<td>ODBC Firebird</td>
<td>1495</td>
<td>18056</td>
<td>13485</td>
<td>17731</td>
</tr>
<tr>
<td>ZEOS Firebird</td>
<td>10452</td>
<td>62851</td>
<td>22003</td>
<td>63708</td>
</tr>
<tr>
<td>FireDAC Firebird</td>
<td>18147</td>
<td>46877</td>
<td>18922</td>
<td>46353</td>
</tr>
</tbody>
</table>
Due to its ACID implementation, SQLite3 process on file waits for the hard-disk to have finished flushing its data, therefore it is the reason why it is slower than other engines at individual row insertion (less than 10 objects per second with a mechanical hard drive instead of a SDD) outside the scope of a transaction.

So if you want to reach the best writing performance in your application with the default engine, you should better use transactions and regroup all writing into services or a BATCH process. Another possibility could be to execute `DB.Synchronous := smOff` and/or `DB.LockingMode := lmExclusive` at SQLite3 engine level before process: in case of power loss at wrong time it may corrupt the database file, but it will increase the rate by a factor of 50 (with hard drive), as stated by the "off" and "off exc" rows of the table - see below (page 222). Note that by default, the FireDAC library set both options, so results above are to be compared with "SQLite3 off exc" rows. In SQLite3 direct mode,

<table>
<thead>
<tr>
<th>Database Type</th>
<th>Scan</th>
<th>Insert</th>
<th>Update</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>UniDAC Oracle</td>
<td>5016</td>
<td>7341</td>
<td>11686</td>
<td>51242</td>
</tr>
<tr>
<td>UniDAC MSSQL</td>
<td>4392</td>
<td>29768</td>
<td>8649</td>
<td>33464</td>
</tr>
<tr>
<td>ODBC DB2</td>
<td>4792</td>
<td>48387</td>
<td>14085</td>
<td>70104</td>
</tr>
<tr>
<td>FireDAC DB2</td>
<td>4452</td>
<td>48635</td>
<td>11014</td>
<td>52781</td>
</tr>
<tr>
<td>ZEOS PostgreSQL</td>
<td>4196</td>
<td>31409</td>
<td>9689</td>
<td>41225</td>
</tr>
<tr>
<td>ODBC PostgreSQL</td>
<td>4068</td>
<td>26262</td>
<td>5130</td>
<td>30435</td>
</tr>
<tr>
<td>FireDAC PostgreSQL</td>
<td>4181</td>
<td>26635</td>
<td>10111</td>
<td>36483</td>
</tr>
<tr>
<td>UniDAC PostgreSQL</td>
<td>2705</td>
<td>18563</td>
<td>4442</td>
<td>28337</td>
</tr>
<tr>
<td>ODBC MySQL</td>
<td>3160</td>
<td>38309</td>
<td>10856</td>
<td>47630</td>
</tr>
<tr>
<td>ZEOS MySQL</td>
<td>3426</td>
<td>34037</td>
<td>12217</td>
<td>40186</td>
</tr>
<tr>
<td>FireDAC MySQL</td>
<td>3078</td>
<td>43053</td>
<td>10955</td>
<td>45781</td>
</tr>
<tr>
<td>UniDAC MySQL</td>
<td>3119</td>
<td>27772</td>
<td>11246</td>
<td>33288</td>
</tr>
</tbody>
</table>
BATCH process benefits of multi-INSERT statements (just like external databases): it explains why BatchAdd() is faster than plain Add(), even in the slowest and safest "file full" mode.

For our direct Oracle access SynDBOracle.pas unit, and for SynDBZeo.pas or SynDBFireDAC.pas (known as Array DML in FireDAC/AnyDAC) libraries, BATCH process benefits of the array binding feature a lot.

For most engines, our ORM kernel is able to generate the appropriate SQL statement for speeding up bulk insertion. For instance:
- SQLite3, MySQL, PostgreSQL, MSSQL 2008, DB2, MySQL or NexusDB handle INSERT statements with multiple INSERT INTO .. VALUES (...),(...),(...),...
- Oracle handles INSERT INTO .. INTO .. SELECT 1 FROM DUAL (weird syntax, isn't it?)
- Firebird implements EXECUTE BLOCK.

As a result, some engines show a nice speed boost when BatchAdd() is used. Even SQLite3 is faster when used as external engine, in respect to direct execution! This feature is at ORM/SQL level, so it benefits to any external database library. Of course, if a given library has a better implementation pattern (e.g. our direct Oracle, Zeos or FireDAC with native array binding), it is used instead.

MongoDB bulk insertion has been implemented, which shows an amazing speed increase in Batch mode. Depending on the MongoDB write concern mode, insertion speed can be very high: by default, every write process will be acknowledged by the server, but you can by-pass this request if you set the wcUnacknowledged mode - note that in this case, any error (e.g. an unique field duplicated value) will never be notified, so it should not be used in production, unless you need this feature to quickly populate a database, or consolidate some data as fast as possible.

### 7.1.3.3. Reading speed

Now the same data is retrieved via the ORM layer:
- 'By one' states that one object is read per call (ORM generates a SELECT * FROM table WHERE ID=? for Client.Retrieve() method);
- 'All *' is when all 5000 objects are read in a single call (i.e. running SELECT * FROM table from a FillPrepare() method call), either forced to use the virtual table layer, or with direct static call.

Here are some reading speed values, in objects/second:

<table>
<thead>
<tr>
<th></th>
<th>By one</th>
<th>All Virtual</th>
<th>All Direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite3 (file full)</td>
<td>127284</td>
<td>558721</td>
<td>550842</td>
</tr>
<tr>
<td>SQLite3 (file off)</td>
<td>126896</td>
<td>549450</td>
<td>526149</td>
</tr>
<tr>
<td>SQLite3 (file off exc)</td>
<td>128077</td>
<td>557537</td>
<td>535905</td>
</tr>
<tr>
<td>SQLite3 (mem)</td>
<td>127106</td>
<td>557537</td>
<td>563316</td>
</tr>
<tr>
<td>TObjectList (static)</td>
<td>300012</td>
<td>912408</td>
<td>913742</td>
</tr>
<tr>
<td>TObjectList (virtual)</td>
<td>303287</td>
<td>402706</td>
<td>866551</td>
</tr>
<tr>
<td>SQLite3 (ext full)</td>
<td>135380</td>
<td>267436</td>
<td>553158</td>
</tr>
<tr>
<td>SQLite3 (ext off)</td>
<td>133696</td>
<td>262977</td>
<td>543065</td>
</tr>
<tr>
<td>SQLite3 (ext off exc)</td>
<td>134698</td>
<td>264186</td>
<td>558596</td>
</tr>
<tr>
<td>Database Type</td>
<td>Connections</td>
<td>Transactions</td>
<td>Slow Queries</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>SQLite3 (ext mem)</td>
<td>137487</td>
<td>259713</td>
<td>557475</td>
</tr>
<tr>
<td>WinHTTP SQLite3</td>
<td>2198</td>
<td>209231</td>
<td>340460</td>
</tr>
<tr>
<td>Sockets SQLite3</td>
<td>8524</td>
<td>210260</td>
<td>387687</td>
</tr>
<tr>
<td>MongoDB (ack)</td>
<td>8002</td>
<td>262353</td>
<td>271268</td>
</tr>
<tr>
<td>MongoDB (no ack)</td>
<td>8234</td>
<td>272079</td>
<td>274582</td>
</tr>
<tr>
<td>ODBC SQLite3</td>
<td>19461</td>
<td>136600</td>
<td>201280</td>
</tr>
<tr>
<td>ZEOS SQLite3</td>
<td>33541</td>
<td>200835</td>
<td>306955</td>
</tr>
<tr>
<td>FireDAC SQLite3</td>
<td>7683</td>
<td>83532</td>
<td>112470</td>
</tr>
<tr>
<td>UniDAC SQLite3</td>
<td>2522</td>
<td>74030</td>
<td>96420</td>
</tr>
<tr>
<td>ODBC Firebird</td>
<td>3446</td>
<td>69607</td>
<td>97585</td>
</tr>
<tr>
<td>ZEOS Firebird</td>
<td>20296</td>
<td>114676</td>
<td>117210</td>
</tr>
<tr>
<td>FireDAC Firebird</td>
<td>2376</td>
<td>46276</td>
<td>56269</td>
</tr>
<tr>
<td>UniDAC Firebird</td>
<td>2189</td>
<td>66886</td>
<td>88102</td>
</tr>
<tr>
<td>Jet</td>
<td>2640</td>
<td>166112</td>
<td>258277</td>
</tr>
<tr>
<td>NexusDB</td>
<td>1413</td>
<td>120845</td>
<td>208246</td>
</tr>
<tr>
<td>Oracle</td>
<td>1558</td>
<td>120977</td>
<td>159861</td>
</tr>
<tr>
<td>ZEOS Oracle</td>
<td>1420</td>
<td>110367</td>
<td>137982</td>
</tr>
<tr>
<td>ODBC Oracle</td>
<td>1620</td>
<td>43441</td>
<td>45764</td>
</tr>
<tr>
<td>FireDAC Oracle</td>
<td>1231</td>
<td>42149</td>
<td>54795</td>
</tr>
<tr>
<td>UniDAC Oracle</td>
<td>688</td>
<td>27083</td>
<td>30093</td>
</tr>
<tr>
<td>BDE Oracle</td>
<td>860</td>
<td>3870</td>
<td>4036</td>
</tr>
<tr>
<td>MSSQL local</td>
<td>10135</td>
<td>210837</td>
<td>437905</td>
</tr>
<tr>
<td>ODBC MSSQL</td>
<td>12458</td>
<td>147544</td>
<td>256502</td>
</tr>
<tr>
<td>FireDAC MSSQL</td>
<td>3776</td>
<td>72123</td>
<td>94091</td>
</tr>
<tr>
<td>UniDAC MSSQL</td>
<td>2505</td>
<td>93231</td>
<td>135932</td>
</tr>
<tr>
<td>ODBC DB2</td>
<td>7649</td>
<td>84880</td>
<td>124486</td>
</tr>
<tr>
<td>FireDAC DB2</td>
<td>3155</td>
<td>71456</td>
<td>88264</td>
</tr>
<tr>
<td>ZEOS PostgreSQL</td>
<td>8833</td>
<td>158760</td>
<td>223583</td>
</tr>
<tr>
<td>ODBC PostgreSQL</td>
<td>10361</td>
<td>85680</td>
<td>120913</td>
</tr>
<tr>
<td>FireDAC PostgreSQL</td>
<td>2261</td>
<td>58252</td>
<td>79002</td>
</tr>
</tbody>
</table>
The SQLite3 layer gives amazing reading results, which makes it a perfect fit for most typical ORM use. When running with DB.LockingMode := lmExclusive defined (i.e. "off exc" rows), reading speed is very high, and benefits from exclusive access to the database file - see below (page 222). External database access is only required when data is expected to be shared with other processes, or for better scaling; e.g. for physical n-Tier installation with dedicated database server(s).

In the above table, it appears that all libraries based on DB.pas are slower than the others for reading speed. In fact, TDataSet sounds to be a real bottleneck, due to its internal data marshalling. Even FireDAC, which is known to be very optimized for speed, is limited by the TDataSet structure. Our direct classes, or even ZEOS/ZDBC performs better, since they are able to output JSON content with no additional marshalling, via a dedicated ColumnsToJSON() method.

For both writing and reading, TObjectList / TSQLRestStorageInMemory engine gives impressive results, but has the weakness of being in-memory, so it is not ACID by design, and the data has to fit in memory. Note that indexes are available for IDs and stored AS_UNIQUE properties.

As a consequence, search of non-unique values may be slow: the engine has to loop through all rows of data. But for unique values (defined as stored AS_UNIQUE), both insertion and search speed is awesome, due to its optimized O(1) hash algorithm - see the following benchmark, especially the "By name" row for "TObjectList" columns, which correspond to a search of an unique RawUTF8 property value via this hashing method.

---

### Benchmark Results

<table>
<thead>
<tr>
<th>Library</th>
<th>SQLite3 (file full)</th>
<th>SQLite3 (file off)</th>
<th>SQLite3 (mem)</th>
<th>TObjectList (static)</th>
<th>TObjectList (virt.)</th>
<th>SQLite3 (ext file full)</th>
<th>SQLite3 (ext file off)</th>
<th>SQLite3 (ext mem)</th>
<th>Oracle</th>
<th>Jet</th>
</tr>
</thead>
<tbody>
<tr>
<td>UniDAC PostgreSQL</td>
<td>864</td>
<td>86900</td>
<td>122856</td>
<td>10143</td>
<td>65538</td>
<td>82447</td>
<td>2052</td>
<td>171803</td>
<td>245772</td>
<td>3636</td>
</tr>
<tr>
<td>ODBC MySQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZEOS MySQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FireDAC MySQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UniDAC MySQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The above table results were run on a Core 2 duo laptop, so numbers are lower than with the previous tables.

During the tests, internal caching - see below (page 317) and ORM Cache (page 174) - was disabled, so you may expect speed enhancements for real applications, when data is more read than written: for instance, when an object is retrieved from the cache, you achieve more than 1,00,000 read requests per second, whatever database is used.
7.1.3.4. Analysis and use case proposal

When declared as virtual table (via a `VirtualTableRegister` call), you have the full power of SQL (including JOINs) at hand, with incredibly fast CRUD operations: 100,000 requests per second for objects read and write, including serialization and Client-Server communication!

Some providers are first-class citizens to mORMot, like SQLite3, Oracle, MS SQL, PostgreSQL, MySQL or IBM DB2. You can connect to them without the bottleneck of the DB.pas unit, nor any restriction of your Delphi license (a Starter edition is enough).

First of all, SQLite3 is still to be considered, even for a production server. Thanks to mORMot’s architecture and design, this “embedded” database could be used as main database engine for a client-server application with heavy concurrent access - if you have doubts about its scaling abilities, see below (page 335). Here, “embedded” is not restricted to “mobile”, but sounds like a self-contained, zero-configuration proven engine.

The remote access via HTTP gives pretty good results, and in this local benchmark, plain socket client (i.e. TSQLDBSocketConnectionProperties class) gives better results that the WinHTTP API (using TSQLDBWinHTTPConnectionProperties on the client side). But in real use, e.g. over the Internet, the WinHTTP API has been reported as more stable, so may be preferred on production. With a SQLite3 backend, this offers pretty good performance, and the benefit of using standard HTTP for its transport.

Most recognized closed source databases are available:

- Direct access to Oracle gives impressive results in BATCH mode (aka array binding). It may be an obligation if your end-customer stores already its data in such a server, for instance, and want to leverage the licensing cost of its own IT solution. Oracle Express edition is free, but somewhat heavy and limited in terms of data/hardware size (see its licensing terms);
- MS SQL Server, directly accessed via OleDB (or ODBC) gives pretty good timing. A MS SQL Server 2008 R2 Express instance is pretty well integrated with the Windows environment, for a very affordable price (i.e. for free) - the LocalDB (MSI installer) edition is enough to start with, but also with data/hardware size limitation, just like Oracle Express;
- IBM DB2 is another good candidate, and the Express-C ("C" standing for Community) offers a no-charge opportunity to run an industry standard engine, with no restriction on the data size, and somewhat high hardware limitations (16 GB of RAM and 2 CPU cores for the latest 10.5 release) or enterprise-level features;
- We did not include Informix numbers here, since support for this database was provided by an user patch - thanks Esteban Martin for sharing! - and we do not have any such server available here;
- NexusDB may be considered, if you have existing Delphi code and data - but it is less known and recognized as the its commercial competitors.

Open Source databases are worth considering, especially in conjunction with an Open Source framework like mORMot:

- MySQL is the well-known engine used by a lot of web sites, mainly with LAMP (Linux Apache MySQL PHP) configurations. Windows is not the best platform to run it, but it could be a fairly good candidate, especially in its MariaDB fork, which sounds more attractive those days than the official main version, owned by Oracle;
- PostgreSQL is an Enterprise class database, with amazing features among its Open Source alternatives, and really competes with commercial solutions. Even under Windows, we think it is easy to install and administrate, and uses less resource than the other commercial engines.
- Firebird gave pretty consistent timing, when accessed via Zeos/ZDBC. We show here the embedded version, but the server edition is worth considering, since a lot of Delphi programmers are skilled with this free alternative to Interbase;
- *MongoDB* appears as a serious competitor to SQL databases, with the potential benefit of horizontal scaling and installation/administration ease - performance is very high, and its document-based storage fits perfectly with *mORMot's* advanced ORM features like *Shared nothing architecture (or sharding)* (page 155).

To access those databases, OleDB, ODBC or ZDBC providers may also be used, with direct access. *mORMot* is a very open-minded rodent: you can use any *DB.pas* provider, e.g. *FireDAC*, *UniDAC*, *DBExpress*, *NexusDB* or even the *BDE*, but with the additional layer introduced by using a TDataSet instance, at reading.

Therefore, the typical use may be the following:

**Table 1**

<table>
<thead>
<tr>
<th>Database</th>
<th>Use case</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal SQLite3 file</td>
<td>Created by default. General safe data handling, with amazing speed in &quot;off exc&quot; mode</td>
</tr>
<tr>
<td>internal SQLite3 in-memory</td>
<td>Created with ':memory:' file name. Fast data handling with no persistence (e.g. for testing or temporary storage)</td>
</tr>
<tr>
<td>TObjectList static</td>
<td>Created with StaticDataCreate. Best possible performance for small amount of data, without ACID nor SQL</td>
</tr>
<tr>
<td>TObjectList virtual</td>
<td>Created with VirtualTableRegister. Best possible performance for SQL over small amount of data (or even unlimited amount under Win64), if ACID is not required nor complex SQL</td>
</tr>
<tr>
<td>external SQLite3 file</td>
<td>Created with VirtualTableExternalRegister. External back-end, e.g. for disk spanning</td>
</tr>
<tr>
<td>external Oracle / MS SQL / DB2 / PostgreSQL / MySQL / Informix / Firebird</td>
<td>Created with VirtualTableExternalRegister. Fast, secure and industry standard back-ends; data can be shared outside <em>mORMot</em></td>
</tr>
<tr>
<td>external NexusDB</td>
<td>Created with VirtualTableExternalRegister. The free embedded version let the whole engine be included within your executable, and use any existing code, but <em>SQLite3</em> sounds like a better option</td>
</tr>
<tr>
<td>external Jet/MSAccess</td>
<td>Created with VirtualTableExternalRegister. Could be used as a data exchange format (e.g. with Office applications)</td>
</tr>
<tr>
<td>external Zeos</td>
<td>Created with VirtualTableExternalRegister. Allow access to several external engines, with direct Zeos/ZDBC access which will by-pass the DB.pas unit and its TDataSet bottleneck - and we will also prefer an active Open Source project!</td>
</tr>
</tbody>
</table>
external FireDAC/UniDAC
Created with VirtualTableExternalRegister
Allow access to several external engines, including the DB.pas unit and its TDataSet bottleneck

eexternal MongoDB
Created with StaticMongoDBRegister()
High-speed document-based storage, with horizontal scaling and advanced query abilities of nested sub-documents

Whatever database back-end is used, don't forget that mORMot design will allow you to switch from one library to another, just by changing a TSQLDatabaseConnectionProperties class type. And note that you can mix external engines, on purpose: you are not tied to one single engine, but the database access can be tuned for each ORM table, according to your project needs.
7.2. SQLite3 implementation

Beginning with the revision 1.15 of the framework, the SQLite3 engine itself has been separated from our mORMotSQLite3.pas unit, and defined as a stand-alone unit named SynSQLite3.pas. See SDD # DI-2.2.1.

It can be used therefore:
- Either stand-alone with direct access of all its features, even using its lowest-level C API, via SynSQLite3.pas - but you won't be able to switch to another database engine easily;
- Or stand-alone with high-level SQL access, using our SynDB.pas generic access classes, via SynDBSQLite3.pas - so you will be able to change to any other database engine (e.g. MS SQL, PostgreSQL, MySQL or Oracle) when needed;
- Or Client-Server based access with all our ORM features - see mORMotSQLite3.pas.

We'll define here some highlights specific to our own implementation of the SQLite3 engine, and let you consult the official documentation of this great Open Source project at http://sqlite.org/ for general information about its common features.

7.2.1. Statically linked or using external dll

Since revision 1.18 of the framework, our SynSQLite3.pas unit is able to access the SQLite3 engine in two ways:
- Either *statically linked* within the project executable;
- Or from an external sqlite3.dll library file.

The SQLite3 APIs and constants are defined in SynSQLite3.pas, and accessible via a TSQLite3Library class definition. It defines a global sqlite3 variable as such:

```var
sqlite3: TSQLite3Library;
```

To use the SQLite3 engine, an instance of TSQLite3Library class shall be assigned to this global variable. Then all mORMot's calls will be made through it, calling e.g. sqlite3.open() instead of sqlite3_open().

There are two implementation classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Unit</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLite3LibraryStatic</td>
<td>SynSQLite3Static.pas</td>
<td>Statically linked engine (sqlite3.obj/sqlite3.o within the .exe)</td>
</tr>
<tr>
<td>TSQLite3LibraryDynamic</td>
<td>SynSQLite3.pas</td>
<td>Instantiate an external sqlite3.dll instance</td>
</tr>
</tbody>
</table>
Referring to SynSQLite3Static.pas in the uses clause of your project is enough to link the .obj/.o engine into your executable.

**Warning - breaking change:** before version 1.18 of the framework, link of static .obj was forced - so you must now add a reference to SynSQLite3Static in your project uses clause to work as expected.

In order to use an external sqlite3.dll library, you have to set the global sqlite3 variable as such:

```pascal
FreeAndNil(sqlite3); // release any previous instance (e.g. static)
sqlite3 := TSQLite3LibraryDynamic.Create;
```

Of course, FreeAndNil(sqlite3) is not mandatory, and should be necessary only to avoid any memory leak if another SQLite3 engine instance was allocated (may be the case if SynSQLite3Static is referred somewhere in your project's units).

Here are some benchmarks, compiled with Delphi XE3, run in a 32-bit project, using either the static bcc-compiled engine, or an external sqlite3.dll, compiled via MinGW or Visual C++.

### 7.2.1.1. Static bcc-compiled .obj
First of all, our version included with SynSQLite3Static.pas unit, is to be benchmarked.

#### Writing speed

<table>
<thead>
<tr>
<th></th>
<th>Direct</th>
<th>Batch</th>
<th>Trans</th>
<th>Batch Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite3 (file full)</td>
<td>477</td>
<td>389</td>
<td>97633</td>
<td>122865</td>
</tr>
<tr>
<td>SQLite3 (file off)</td>
<td>868</td>
<td>869</td>
<td>96827</td>
<td>125862</td>
</tr>
<tr>
<td>SQLite3 (mem)</td>
<td>84642</td>
<td>108624</td>
<td>104947</td>
<td>135105</td>
</tr>
<tr>
<td>TObjectList (static)</td>
<td>338478</td>
<td>575373</td>
<td>337336</td>
<td>572147</td>
</tr>
<tr>
<td>TObjectList (virtual)</td>
<td>338180</td>
<td>554446</td>
<td>331873</td>
<td>575837</td>
</tr>
<tr>
<td>SQLite3 (ext full)</td>
<td>486</td>
<td>496</td>
<td>101413</td>
<td>7011</td>
</tr>
<tr>
<td>SQLite3 (ext off)</td>
<td>799</td>
<td>303</td>
<td>105402</td>
<td>135109</td>
</tr>
<tr>
<td>SQLite3 (ext mem)</td>
<td>93893</td>
<td>129550</td>
<td>109027</td>
<td>152811</td>
</tr>
</tbody>
</table>

#### Reading speed

<table>
<thead>
<tr>
<th></th>
<th>By one</th>
<th>All Virtual</th>
<th>All Direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite3 (file full)</td>
<td>26924</td>
<td>494559</td>
<td>500200</td>
</tr>
<tr>
<td>SQLite3 (file off)</td>
<td>27750</td>
<td>496919</td>
<td>502714</td>
</tr>
<tr>
<td>SQLite3 (mem)</td>
<td>124402</td>
<td>444404</td>
<td>495392</td>
</tr>
<tr>
<td>TObjectList (static)</td>
<td>332778</td>
<td>907605</td>
<td>910249</td>
</tr>
<tr>
<td>TObjectList (virtual)</td>
<td>331038</td>
<td>404891</td>
<td>905961</td>
</tr>
</tbody>
</table>
Good old Borland C++ builder produces some efficient code here. Those numbers are very good, when compared to the other two options. Probably, using FastMM4 as memory manager and tuned compilation options does make sense.

### 7.2.1.2. Official MinGW-compiled sqlite3.dll

Here we used the official sqlite3.dll library, as published in the [http://sqlite.org](http://sqlite.org) web site, and compiled with the MinGW/GCC compiler.

#### Writing speed

<table>
<thead>
<tr>
<th></th>
<th>Direct</th>
<th>Batch</th>
<th>Trans</th>
<th>Batch Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite3 (file full)</td>
<td>418</td>
<td>503</td>
<td>86322</td>
<td>119420</td>
</tr>
<tr>
<td>SQLite3 (file off)</td>
<td>918</td>
<td>873</td>
<td>93196</td>
<td>127317</td>
</tr>
<tr>
<td>SQLite3 (mem)</td>
<td>83108</td>
<td>106951</td>
<td>99892</td>
<td>138003</td>
</tr>
<tr>
<td>TObjectList (static)</td>
<td>320204</td>
<td>573723</td>
<td>324696</td>
<td>547465</td>
</tr>
<tr>
<td>TObjectList (virtual)</td>
<td>323247</td>
<td>563697</td>
<td>324443</td>
<td>564716</td>
</tr>
<tr>
<td>SQLite3 (ext full)</td>
<td>501</td>
<td>410</td>
<td>100152</td>
<td>133679</td>
</tr>
<tr>
<td>SQLite3 (ext off)</td>
<td>913</td>
<td>438</td>
<td>102806</td>
<td>135545</td>
</tr>
<tr>
<td>SQLite3 (ext mem)</td>
<td>96028</td>
<td>122798</td>
<td>108363</td>
<td>150920</td>
</tr>
</tbody>
</table>

#### Reading speed

<table>
<thead>
<tr>
<th></th>
<th>By one</th>
<th>All Virtual</th>
<th>All Direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite3 (file full)</td>
<td>26883</td>
<td>473529</td>
<td>438904</td>
</tr>
<tr>
<td>SQLite3 (file off)</td>
<td>27729</td>
<td>472188</td>
<td>451304</td>
</tr>
<tr>
<td>SQLite3 (mem)</td>
<td>116550</td>
<td>459432</td>
<td>457959</td>
</tr>
<tr>
<td>TObjectList (static)</td>
<td>318248</td>
<td>891265</td>
<td>905469</td>
</tr>
<tr>
<td>TObjectList (virtual)</td>
<td>327739</td>
<td>359040</td>
<td>892697</td>
</tr>
<tr>
<td>SQLite3 (ext full)</td>
<td>127346</td>
<td>180812</td>
<td>370288</td>
</tr>
<tr>
<td>SQLite3 (ext off)</td>
<td>127749</td>
<td>227759</td>
<td>438096</td>
</tr>
<tr>
<td>SQLite3 (ext mem)</td>
<td>129792</td>
<td>224386</td>
<td>436338</td>
</tr>
</tbody>
</table>
7.2.1.3. Visual C++ compiled sqlite3.dll

The Open Source wxsqlite project provides a sqlite3.dll library, compiled with Visual C++, and including RC4 and AES 128/256 encryption (better than the basic encryption implemented in SynSQLite3Static.pas) - not available in the official library.

See http://sourceforge.net/projects/wxcode/files/Components/wxSQLite3.. to download the corresponding source code, and compiled .dll.

### Writing speed

<table>
<thead>
<tr>
<th></th>
<th>Direct</th>
<th>Batch</th>
<th>Trans</th>
<th>Batch Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite3 (file full)</td>
<td>470</td>
<td>498</td>
<td>93801</td>
<td>112170</td>
</tr>
<tr>
<td>SQLite3 (file off)</td>
<td>886</td>
<td>819</td>
<td>90298</td>
<td>132883</td>
</tr>
<tr>
<td>SQLite3 (mem)</td>
<td>86897</td>
<td>110287</td>
<td>105207</td>
<td>140896</td>
</tr>
<tr>
<td>TObjectList (static)</td>
<td>332005</td>
<td>596445</td>
<td>321357</td>
<td>570776</td>
</tr>
<tr>
<td>TObjectList (virtual)</td>
<td>327225</td>
<td>585000</td>
<td>329272</td>
<td>579240</td>
</tr>
<tr>
<td>SQLite3 (ext full)</td>
<td>459</td>
<td>503</td>
<td>91086</td>
<td>140599</td>
</tr>
<tr>
<td>SQLite3 (ext off)</td>
<td>501</td>
<td>519</td>
<td>110338</td>
<td>150394</td>
</tr>
<tr>
<td>SQLite3 (ext mem)</td>
<td>98112</td>
<td>133276</td>
<td>117346</td>
<td>158634</td>
</tr>
</tbody>
</table>

### Reading speed

<table>
<thead>
<tr>
<th></th>
<th>By one</th>
<th>All Virtual</th>
<th>All Direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite3 (file full)</td>
<td>28527</td>
<td>516689</td>
<td>521159</td>
</tr>
<tr>
<td>SQLite3 (file off)</td>
<td>28927</td>
<td>513769</td>
<td>519156</td>
</tr>
<tr>
<td>SQLite3 (mem)</td>
<td>127740</td>
<td>529100</td>
<td>523176</td>
</tr>
<tr>
<td>TObjectList (static)</td>
<td>335053</td>
<td>869262</td>
<td>879352</td>
</tr>
<tr>
<td>TObjectList (virtual)</td>
<td>334739</td>
<td>410374</td>
<td>885269</td>
</tr>
<tr>
<td>SQLite3 (ext full)</td>
<td>132594</td>
<td>258371</td>
<td>506277</td>
</tr>
<tr>
<td>SQLite3 (ext off)</td>
<td>138159</td>
<td>260892</td>
<td>507717</td>
</tr>
<tr>
<td>SQLite3 (ext mem)</td>
<td>139567</td>
<td>254919</td>
<td>516208</td>
</tr>
</tbody>
</table>

Under Windows, the Visual C++ compiler gives very good results. It is a bit faster than the other two, despite a somewhat less efficient virtual table process.

As a conclusion, our SynSQLite3Static.pas statically linked implementation sounds like the best overall approach for Windows 32-bit: best speed for virtual tables (which is the core of our ORM), and no dll hell. No library to deploy and copy, everything is embedded in the project executable, ready to run as expected. External sqlite3.dll will be used for cross-platform support, and when targeting
64-bit Windows applications.

### 7.2.2. Prepared statement

In order to speed up the time spent in the SQLite3 engine (it may be useful for high-end servers), the framework is able to natively handle prepared SQL statements.

Starting with version 1.12 of the framework, we added an internal SQL statement cache in the database access, available for all SQL request. Previously, only the one-record SQL SELECT * FROM ... WHERE RowID=... was prepared (used e.g. for the TSQLRest. Retrieve method).

That is, if a previous SQL statement is run with some given parameters, a prepared version, available in cache, is used, and new parameters are bounded to it before the execution by SQLite3.

In some cases, it can speed the SQLite3 process a lot. From our profiling, prepared statements make common requests (i.e. select / insert / update on one row) at least two times faster, on an in-memory database (':memory:' specified as file name).

In order to use this statement caching, any SQL statements must have the parameters to be surrounded with ':(' and ')':. The SQL format was indeed enhanced by adding an optional way of marking parameters inside the SQL request, to enforce statement preparing and caching.

Therefore, there are now two ways of writing the same SQL request:

Write the SQL statement as usual:

```sql
SELECT * FROM TABLE WHERE ID=10;
```

In this case, the SQL will be parsed by the SQLite3 engine, a statement will be compiled, then run.

Use the new optional markers to identify the changing parameter:

```sql
SELECT * FROM TABLE WHERE ID=:(10):;
```

in this case, any matching already prepared statement will be re-used for direct run.

In the later case, an internal pool of prepared TSQLRequest statements is maintained. The generic SQL code used for the matching will be this one:

```sql
SELECT * FROM TABLE WHERE ID=?;
```

and the integer value 10 will be bounded to the prepared statement before execution.

Example of possible inlined values are (note double " quotes are allowed for the text parameters, whereas SQL statement should only use single ' quotes):

```sql
:(1234): :(12.34): :(12E-34): :'text': :('It''s great'):
```

All internal SQL statement generated by the ORM are now using this new parameter syntax.

For instance, here is how an object deletion is implemented for the SQLite3 engine:

```pascal
function TSQLRestServerDB.EngineDelete(Table: TSQLRecordClass; ID: TID): boolean;
begin
  if Assigned(OnUpdateEvent) then
    OnUpdateEvent(self,seDelete,Table,ID); // notify BEFORE deletion

  result := ExecuteFmt('DELETE FROM % WHERE RowID=:(%):;',[Table.SQLTableName,ID]);
end;
```

Using :(%): will let the DELETE FROM table_name WHERE RowID=? statement be prepared and reused between calls.

In your code, you should better use, for instance:
aName := OneFieldValue(TSQLMyRecord,'Name', 'ID:=:(%):',[aID]);

or even easier

aName := OneFieldValue(TSQLMyRecord,'Name', 'ID=?',[],[aID]);

instead of

aName := OneFieldValue(TSQLMyRecord,'Name', 'ID=%',[aID]);

or instead of a plain

aName := OneFieldValue(TSQLMyRecord,'Name', 'ID='+Int32ToUtf8(aID));

In fact, from your client code, you may not use directly the :(...) expression in your request, but will rather use the overloaded TSQLRecord.Create, TSQLRecord.FillPrepare, TSQLRecord.CreateAndFillPrepare, TSQLRest.OneFieldValue, TSQLRest.MultiFieldValue, TQLRestClient.ExecuteFmt and TSQLRestClient.ListFmt methods, available since revision 1.15 of the framework, which will accept both '%' and '?' characters in the SQL WHERE format text, in-lining '?' parameters with proper :(...) encoding and quoting the RawUTF8/strings parameters on purpose.

I found out that this SQL format enhancement is much easier to use (and faster) in the Delphi code than using parameters by name or by index, like in this classic VCL code:

```delphi
SQL.Text := 'SELECT Name FROM Table WHERE ID=:Index';
SQL.ParamByName('Index').AsInteger := aID;
```

At a lowest-level, in-lining the bounds values inside the statement enabled better serialization in a Client-Server architecture, and made caching easier on the Server side: the whole SQL query contains all parameters within one unique RawUTF8 value, and can be therefore directly compared to the cached entries. As such, our framework is able to handle prepared statements without keeping bound parameters separated from the main SQL text.

It is also worth noting that external databases (see next paragraph) will also benefit from this statement preparation. Inlined values will be bound separately to the external SQL statement, to achieve the best speed possible.

### 7.2.3. R-Tree inclusion

Since the 2010-06-25 source code repository update, the RTREE extension is now compiled by default within all supplied .obj files.

An R-Tree is a special index that is designed for doing range queries. R-Trees are most commonly used in geospatial systems where each entry is a rectangle with minimum and maximum X and Y coordinates. Given a query rectangle, an R-Tree is able to quickly find all entries that are contained within the query rectangle or which overlap the query rectangle. This idea is easily extended to three dimensions for use in CAD systems. R-Trees also find use in time-domain range look-ups. For example, suppose a database records the starting and ending times for a large number of events. A R-Tree is able to quickly find all events, for example, that were active at any time during a given time interval, or all events that started during a particular time interval, or all events that both started and ended within a given time interval. And so forth. See [http://www.sqlite.org/rtree.html](http://www.sqlite.org/rtree.html).

A dedicated ORM class, named TSQLRecordRTree, is available to create such tables. It inherits from TSQLRecordVirtual, like the other virtual tables types (e.g. TSQLRecordFTS5).

Any record which inherits from this TSQLRecordRTree class must have only sftFloat (i.e. Delphi double) published properties grouped by pairs, each as minimum- and maximum-value, up to 5 dimensions (i.e. 11 columns, including the ID property). Its ID: TID property must be set before adding a TSQLRecordRTree to the database, e.g. to link an R-Tree representation to a regular
TSQLRecord table containing the main data.

Queries against the ID or the coordinate ranges are almost immediate: so you can e.g. extract some coordinates box from the main regular TSQLRecord table, then use a TSQLRecordRTree-joined query to make the process faster; this is exactly what the TSQLRestClient. RTreeMatch method offers: for instance, running with aMapData. BlobField filled with [-81, -79.6, 35, 36.2] the following lines:

```plaintext
aClient.RTreeMatch(TSQLRecordMapData,'BlobField',TSQLRecordMapBox,
    aMapData.BlobField,ResultID);
```

will execute the following SQL statement:

```plaintext
SELECT MapData.ID From MapData, MapBox WHERE MapData.ID=MapBox.ID
AND minX>=:(-81.0): AND maxX<=:(-79.6): AND minY>=:(35.0): AND :maxY<=36.2):
AND MapBox_in(MapData.BlobField,:('uFFFF0base64encoded-81,-79.6,35,36.2'));
```

The MapBox_in SQL function is registered in TSQLRestServerDB. Create constructor for all TSQLRecordRTree classes of the current database model. Both BlobToCoord and ContainedIn class methods are used to handle the box storage in the BLOB. By default, it will process a raw array of double, with a default box match (that is ContainedIn method will match the simple \( \text{minX} = \ldots \text{maxY} = \ldots \) where clause).

### 7.2.4. FTS3/FTS4/FTS5

FTS3/FTS4/FTS5 are SQLite3 virtual table modules that allow users to perform full-text searches on a set of documents. The most common (and effective) way to describe full-text searches is "what Google, Yahoo and Altavista do with documents placed on the World Wide Web". Users input a term, or series of terms, perhaps connected by a binary operator or grouped together into a phrase, and the full-text query system finds the set of documents that best matches those terms considering the operators and groupings the user has specified.

See [http://www.sqlite.org/fts3.html](http://www.sqlite.org/fts3.html) as reference material about FTS3/FTS4 usage in SQLite3, and [https://www.sqlite.org/fts5.html](https://www.sqlite.org/fts5.html) about FTS5. In short, FTS5 is a new version of FTS4 that includes various fixes and solutions for problems that could not be fixed in FTS4 without sacrificing backwards compatibility.

Since recent versions of the framework, the sqlite3.obj/.o static file available with the distribution includes the FTS3/FTS4/FTS5 engines (also on other platforms with FPC).

#### 7.2.4.1. Dedicated FTS3/FTS4/FTS5 record type

In order to allow easy use of the FTS feature, some types have been defined:

- TSQLRecordFTS3 to create a FTS3 table with default "simple" stemming;
- TSQLRecordFTS3Porter to create a FTS3 table using the Porter Stemming algorithm (see below);
- TSQLRecordFTS3Unicode61 to create a FTS3 table using the Unicode61 Stemming algorithm (see below);
- TSQLRecordFTS4 to create a FTS4 table with default "simple" stemming;
- TSQLRecordFTS4Porter to create a FTS4 table using the Porter Stemming algorithm;
- TSQLRecordFTS4Unicode61 to create a FTS4 table using the Unicode61 Stemming;
- TSQLRecordFTS5 to create a FTS5 table with default "simple" stemming;
- TSQLRecordFTS5Porter to create a FTS5 table using the Porter Stemming algorithm;
- TSQLRecordFTS5Unicode61 to create a FTS5 table using the Unicode61 Stemming;

The following graph will detail this class hierarchy:
7.2.4.2. Stemming

The "stemming" algorithm - see http://sqlite.org/fts3.html#tokenizer - is the way the english text is parsed for creating the word index from raw text.

The simple (default) tokenizer extracts tokens from a document or basic FTS full-text query according to the following rules:

- A term is a contiguous sequence of eligible characters, where eligible characters are all alphanumeric characters, the "_" character, and all characters with UTF code-points greater than or equal to 128. All other characters are discarded when splitting a document into terms. Their only contribution is to separate adjacent terms.
- All uppercase characters within the ASCII range (UTF code-points less than 128), are transformed to their lowercase equivalents as part of the tokenization process. Thus, full-text queries are case-insensitive when using the simple tokenizer.

For example, when a document containing the text "Right now, they're very frustrated.", the terms extracted from the document and added to the full-text index are, in order, "right now they're very frustrated". Such a document will match a full-text query such as "MATCH 'Frustrated'", as the simple tokenizer transforms the term in the query to lowercase before searching the full-text index.

The Porter Stemming algorithm tokenizer uses the same rules to separate the input document into terms, but as well as folding all terms to lower case it uses the Porter Stemming algorithm to reduce related English language words to a common root. For example, using the same input document as in the paragraph above, the porter tokenizer extracts the following tokens: "right now the very frustrated". Even though some of these terms are not even English words, in some cases using them to build the full-text index is more useful than the more intelligible output produced by the simple
tokenizer. Using the porter tokenizer, the document not only matches full-text queries such as "MATCH 'Frustrated'", but also queries such as "MATCH 'Frustration'", as the term "Frustration" is reduced by the Porter stemmer algorithm to "frustrat" - just as "Frustrated" is. So, when using the porter tokenizer, FTS is able to find not just exact matches for queried terms, but matches against similar English language terms. For more information on the Porter Stemmer algorithm, please refer to the [http://tartarus.org/~martin/PorterStemmer.](http://tartarus.org/~martin/PorterStemmer).

The Unicode61 Stemming algorithm tokenizer works very much like "simple" except that it does simple unicode case folding according to rules in Unicode Version 6.1 and it recognizes unicode space and punctuation characters and uses those to separate tokens. By default, "Unicode61" also removes all diacritics from Latin script characters.

### 7.2.4.3. FTS searches

A good approach is to store your data in a regular TSQLRecord table, then store your text content in a separated FTS table, associated to this TSQLRecordFTS table via its ID / DocID property. Note that for TSQLRecordFTS* types, the ID property was renamed as DocID, which is the internal name for the FTS virtual table definition of its unique integer key ID property.

For example (extracted from the regression test code), you can define this new class:

```delphi
TSQLFTSTest = class(TSQLRecordFTS5)
private
  fSubject: RawUTF8;
  fBody: RawUTF8;
published
  property Subject: RawUTF8 read fSubject write fSubject;
  property Body: RawUTF8 read fBody write fBody;
end;
```

Note that FTS tables must only content UTF-8 text field, that is RawUTF8 (under Delphi 2009 and up, you could also use the Unicode string type, which is mapped as a UTF-8 text field for the SQLite3 engine).

Then you can add some Body/Subject content to this FTS table, just like any regular TSQLRecord content, via the ORM feature of the framework:

```delphi
FTS := TSQLFTSTest.Create;
try
  Check(aClient.TransactionBegin(TSQLFTSTest)); // MUCH faster with this
  for i := StartID to StartID+COUNT-1 do
  begin
    FTS.DocID := IntArray[i];
    FTS.Subject := aClient.OneFieldValue(TSQLRecordPeople,'FirstName',FTS.DocID);
    FTS.Body := FTS.Subject+' bodY'+IntToStr(FTS.DocID);
    aClient.Add(FTS,true);
  end;
  aClient.Commit; // Commit must be BEFORE OptimizeFTS3, memory leak otherwise
  Check(FTS.OptimizeFTS3Index(Client.fServer));
end;
```

The steps above are just typical. The only difference with a "standard" ORM approach is that the DocID property must be set before adding the TSQLRecordFTS5 instance: there is no ID automatically created by SQLite, but an ID must be specified in order to link the FTS record to the original TSQLRecordPeople row, from its ID.

To support full-text queries, FTS maintains an inverted index that maps from each unique term or word that appears in the dataset to the locations in which it appears within the table contents. The dedicated OptimizeFTS3Index method is called to merge all existing index b-trees into a single large b-tree containing the entire index - this method will work with FTS3, FTS4 and FTS5 classes, whatever
its name states. This can be an expensive operation, but may speed up future queries: you should not
call this method after every modification of the FTS tables, but after some text has been added.

Then the FTS search query will use the custom FTSMatch method:

```check(aClient.FTSMatch(TSQLFTSTest,'Subject MATCH ''salVador1''',IntResult));```

The matching IDs are stored in the IntResult integer dynamic array. Note that you can use a regular
SQL query instead. Use of the FTSMatch method is not mandatory: in fact, it is just a wrapper around
the OneFieldValues method, just using the "neutral" RowID column name for the results:

```function TSQLRest.FTSMatch(Table: TSQLRecordFTS3Class;
const WhereClause: RawUTF8; var DocID: TIntegerDynArray): boolean;
begin // FTS3 tables do not have any ID, but RowID or DocID
result := OneFieldValues(Table,'RowID',WhereClause,DocID);
end;```

An overloaded FTSMatch method has been defined, and will handle detailed matching information,
able to use a ranking algorithm. With this method, the results will be sorted by relevance:

```Check(aClient.FTSMatch(TSQLFTSTest,'body1*',IntResult,[1,0.5]));```

This method expects some additional constant parameters for weighting each FTS table column (there
must be the same number of PerFieldWeight parameters as there are columns in the
TSQLRecordFTSS table). In the above sample code, the Subject field will have a weight of 1.0, and the
Body will be weighted as 0.5, i.e. any match in the 'body' column content will be ranked twice less than
any match in the 'subject', which is probably of higher density.

The above query will call the following SQL statement:

```SELECT RowID FROM FTSTest WHERE FTSTest MATCH 'body1*' ORDER BY rank(matchinfo(FTSTest),1.0,0.5) DESC```

The rank internal SQL function has been implemented in Delphi, following the guidelines of the official
SQLite3 documentation - as available from their internet web site at
http://www.sqlite.org/fts3.html#appendix_a.. - to implement the most efficient way of implementing
ranking. It will return the RowID of documents that match the full-text query sorted from most to least
relevant. When calculating relevance, query term instances in the 'subject' column are given twice the
weighting of those in the 'body' column.

### 7.2.4.4. FTS4 index tables without content

Just as SQLite3 allows, the framework permits FTS4 to forego storing the text being indexed, letting the
indexed documents be stored in a database table created and managed by the user (an "external
content" FTS4 table).

Because the indexed documents themselves are usually much larger than the full-text index, this
option can be used to achieve significant storage space savings. Contentless FTS4 tables still support
SELECT statements. However, it is an error to attempt to retrieve the value of any table column other
than the docid column. The auxiliary function matchinfo() may be used - so TSQLRest.FTSMatch
method will work as expected, but snippet() and offsets() will cause an exception at execution.

For instance, in sample "30 - MVC Server", we define those two tables:

```TSQLArticle = class(TSQLContent)
private
  fContent: RawUTF8;
  fTitle: RawUTF8;
  fAbstract: RawUTF8;
  fPublishedMonth: Integer;
  fTags: TIntegerDynArray;```
And we initialized the database model to let all data be stored only in TSQLArticle, not in TSQLArticleSearch, using an "external content" FTS4 table to index the text from the selected Title, Abstract and Content fields of TSQLArticle:

```pascal
function CreateModel: TSQLModel;
begin
  result := TSQLModel.Create([TSQLBlogInfo,TSQLAuthor,TSQLTag,TSQLArticle,TSQLComment,TSQLArticleSearch],'blog');
  result.Props[TSQLArticleSearch].FTS4WithoutContent(TSQLArticle);
  ...
end
```

The TSQLModelRecordProperties.FTS4WithoutContent() will in fact create the needed SQLite3 triggers, to automatically populate the ArticleSearch Full Text indexes when the main Article row changes.

Since this FTS4 feature is specific to SQLite3, and triggers do not work on virtual tables (by now), this method won't do anything if the TSQLArticleSearch or TSQLArticle are on an external database - see below (page 239). Both need to be stored in the main SQLite3 DB.

In the 30 - MVC Server sample, the search will be performed as such:

```pascal
if scop^.getAsRawUTF8('match',match) and fHasFTS then begin
  if scop^.getAsDouble('lastrank',rank) then
    whereClause := 'and rank<?';
  whereClause := 'join (select docid,rank(matchinfo(ArticleSearch),1.0,0.7,0.5) as rank '
    + 'from ArticleSearch where ArticleSearch match ? ' + whereClause + '
    'order by rank desc limit 100) as r on (r.docid=Article.id)';
  articles := RestModel.RetrieveDocVariantArray( TSQLArticle,'',whereClause,[match,rank],
    'id,title,tags,author,authorname,createdat,abstract,contenthtml,rank');
```

In the above query expression, the rank() function is used over the detailed FTS4 search statistics returned by matchinfo(), using a 1.0 weight for any match in the Title column, 0.7 for the Abstract column, and 0.5 for Content. The matching articles content is then returned in an articles: TDocVariant array, ready to be rendered on the web page.

### 7.2.5. Column collations

In any database, there is a need to define how column data is to be compared. It is needed for proper search and ordering of the data. This is the purpose of so-called *collations*.

By default, when SQLite compares two strings, it uses a collating sequence or collating function (two words for the same thing) to determine which string is greater or if the two strings are equal. SQLite has three built-in collating functions: BINARY, NOCASE, and RTRIM:
- **BINARY** - Compares string data using `memcmp()`, regardless of text encoding.
- **NOCASE** - The same as binary, except the 26 upper case characters of ASCII are folded to their lower case equivalents before the comparison is performed. Note that only ASCII characters are case folded. Plain SQLite does not attempt to do full Unicode case folding due to the size of the tables required - but you could use mORMot's SYSTEMNOCASE or WIN32CASE/WIN32NOCASE custom collations for enhanced case folding support (see below);
- **RTRIM** - The same as binary, except that trailing space characters are ignored.

In the *mORMot ORM*, we defined some additional kind of collations, via some internal calls to the `sqlite3_create_collation()` API:

<table>
<thead>
<tr>
<th>TSQlFieldType</th>
<th>Default collation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sftAnsiText</td>
<td>NOCASE</td>
</tr>
<tr>
<td>sftUTF8Text</td>
<td>SYSTEMNOCASE, i.e. using <code>UTF8ILComp()</code>, which will ignore Win-1252 Latin accents</td>
</tr>
<tr>
<td>sftEnumerate</td>
<td></td>
</tr>
<tr>
<td>sftSet</td>
<td></td>
</tr>
<tr>
<td>sftInteger</td>
<td></td>
</tr>
<tr>
<td>sftID</td>
<td></td>
</tr>
<tr>
<td>sftTID</td>
<td></td>
</tr>
<tr>
<td>sftRecord</td>
<td></td>
</tr>
<tr>
<td>sftBoolean</td>
<td></td>
</tr>
<tr>
<td>sftFloat</td>
<td></td>
</tr>
<tr>
<td>sftCurrency</td>
<td></td>
</tr>
<tr>
<td>ftTimeLog</td>
<td></td>
</tr>
<tr>
<td>sftModTime</td>
<td></td>
</tr>
<tr>
<td>sftCreateTime</td>
<td></td>
</tr>
<tr>
<td>sftDateTime</td>
<td></td>
</tr>
<tr>
<td>ftDateTimeMS</td>
<td>ISO8601, i.e. decoding the text into a date/time value before comparison</td>
</tr>
<tr>
<td>sftObject</td>
<td></td>
</tr>
<tr>
<td>sftVariant</td>
<td>BINARY, since it is stored as plain JSON content</td>
</tr>
<tr>
<td>sftBlob</td>
<td></td>
</tr>
<tr>
<td>sftBlobDynArray</td>
<td></td>
</tr>
<tr>
<td>sftBlobCustom</td>
<td>BINARY</td>
</tr>
</tbody>
</table>

You can override those default collation schemes by calling `TSQlRecordProperties.SetCustomCollationForAll()` (which will override all fields collation for a given type) or `SetCustomCollation()` method (which will override a given field) in an overridden class procedure `InternalRegisterCustomProperties()` or `InternalDefineModel()`, so that it will be common to all database models, for both client and server, every time the corresponding TSQlRecord is used.

As an alternative, you may call `TSQlModel.SetCustomCollationForAll()` method, which will do it for a given model.

The following collations are therefore available when using SQLite3 within the *mORMot ORM*:

<table>
<thead>
<tr>
<th>Collation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>Default <code>memcmp()</code> comparison</td>
</tr>
</tbody>
</table>
### 7.2.6. REGEXP operator

Our SQLite3 engine can use regular expression within its SQL queries, by enabling the REGEXP operator in addition to standard SQL operators (=  == != <> IS IN LIKE GLOB MATCH). It will use the Open Source PCRE library to perform the queries.

In order to enable the operator, you should include unit SynSQLite3RegEx.pas to your uses clause, and register the RegExp() SQL function to a given SQLite3 database instance, as such:

```pascal
uses SynCommons, mORMot, mORMotSQLite3, SynSQLite3RegEx;

Server := TSQLRestServerDB.Create(Model, 'test.db3');
try
  CreateRegExpFunction(Server.DB.DB);
  with TSQLRecordPeople.CreateAndFillPrepare(Client,
      'FirstName REGEXP ?', 
      [bFinley]) do
    try
      while FillOne do begin
        Check(LastName='Morse');
        Check(IdemPChar(pointer(FirstName),'SAMUEL FINLEY '));
      end;
    finally
      Free;
    end;
  finally
    Server.Free;
end;
```

The above code will execute the following SQL statement (with a prepared parameter for the regular expression itself):

```sql
SELECT * from People WHERE Firstname REGEXP '\bFinley\b';
```
That is, it will find all objects where TSQLRecordPeople.FirstName will contain the 'Finley' word - in a regular expression, \b defines a word boundary search.

In fact, the REGEXP operator is a special syntax for the regexp() user function. No regexp() user function is defined by default and so use of the REGEXP operator will normally result in an error message. Calling CreateRegExFunction() for a given connection will add a SQL function named "regexp()" at run-time, which will be called in order to implement the REGEXP operator.

It will use the statically linked PCRE library as available since Delphi XE, or will rely on the PCRE.pas wrapper unit as published at [http://www.regular-expressions.info/download/TPerlRegEx.zip](http://www.regular-expressions.info/download/TPerlRegEx.zip) for older versions of Delphi.

This unit will call directly the UTF-8 API of the PCRE library, and maintain a per-connection cache of compiled regular expressions to ensure the best performance possible.

### 7.2.7. ACID and speed

As stated above in Data access benchmark (page 199), the default SQLite3 write speed is quite slow, when running on a normal hard drive. By default, the engine will pause after issuing a OS-level write command. This guarantees that the data is written to the disk, and features the ACID properties of the database engine.

ACID is an acronym for "Atomicity Consistency Isolation Durability" properties, which guarantee that database transactions are processed reliably: for instance, in case of a power loss or hardware failure, the data will be saved on disk in a consistent way, with no potential loss of data.

In SQLite3, ACID is implemented by two means at file level:
- **Synchronous writing**: it means that the engine will wait for any written content to be flushed to disk before processing the next request;
- **File locking**: it means that the database file is locked for exclusive use during writing, allowing several processes to access the same database file concurrently.

Changing these default settings can ensure much better writing performance.

#### 7.2.7.1. Synchronous writing

You can overwrite the default ACID behavior by setting the TSQlDataBase.Synchronous property to smOff instead of the default smFull setting. When Synchronous is set to smOff, SQLite continues without syncing as soon as it has handed data off to the operating system. If the application running SQLite crashes, the data will be safe, but the database might become corrupted if the operating system crashes or the computer loses power before that data has been written to the disk surface. On the other hand, some operations are as much as 50 or more times faster with this setting.

When the tests performed during Data access benchmark (page 199) use Synchronous := smOff, "Write one" speed is enhanced from 8-9 rows per second into about 400 rows per second, on a physical hard drive (SSD or NAS drives may not suffer from this delay).

So depending on your application requirements, you may switch Synchronous setting to off.

To change the main SQLite3 engine synchronous parameter, you may code for instance:

```pascal
Client := TSQLRestClientDB.Create(Model,nil,MainDBFileName,TSQLRestServerDB,false,'');
Client.Server.DB.Synchronous := smOff;
```

Note that this setting is common to a whole TSQLDatabase instance, so will affect all tables handled by the TSQLRestServerDB instance.
But if you defined some SQLite3 external tables - see below (page 239), you can define the setting for a particular external connection, for instance:

```pascal
Props := TSQLDBSQLite3ConnectionProperties.Create(DBFileName,'','','');
VirtualTableExternalRegister(Model,TSQLRecordSample,Props,'SampleRecord');
Client := TSQLRestClientDB.Create(Model,nil,MainDBFileName,TSQLRestServerDB,false,'');
TSQLDBSQLite3Connection(Props.MainConnection).Synchronous := smOff;
```

Never forget that you may have several SQLite3 engines within a single mORMot server!

### 7.2.7.2. File locking

You can overwrite the first default ACID behavior by setting the TSQLDataBase.LockingMode property to lmExclusive instead of the default lmNormal setting. When LockingMode is set to lmExclusive, SQLite will lock the database file for exclusive use during the whole session. It will prevent other processes (e.g. database viewer tools) to access the file at the same time, but small write transactions will be much faster, by a factor usually greater than 40. Bigger transactions involving several hundreds/thousands of INSERT won't be accelerated - but individual insertions will have a major speed up - see Data access benchmark (page 199).

To change the main SQLite3 engine locking mode parameter, you may code for instance:

```pascal
Client := TSQLRestClientDB.Create(Model,nil,MainDBFileName,TSQLRestServerDB,false,'');
Client.Server.DB.LockingMode := lmExclusive;
```

Note that this setting is common to a whole TSQLDatabase instance, so will affect all tables handled by the TSQLRestServerDB instance.

But if you defined some SQLite3 external tables - see below (page 239), you can define the setting for a particular external connection, for instance:

```pascal
Props := TSQLDBSQLite3ConnectionProperties.Create(DBFileName,'','','');
VirtualTableExternalRegister(Model,TSQLRecordSample,Props,'SampleRecord');
Client := TSQLRestClientDB.Create(Model,nil,MainDBFileName,TSQLRestServerDB,false,'');
TSQLDBSQLite3Connection(Props.MainConnection).Synchronous := smOff;
```

In fact, exclusive file locking improves the reading speed by a factor of 4 (in case of individual row retrieval). As such, defining LockingMode := lmExclusive without Synchronous := smOff could be of great benefit for a server which purpose is mainly to serve ORM content to clients.

### 7.2.7.3. Performance tuning

By default, the slow but truly ACID setting will be used with mORMot, just as with SQLite3. We do not change this policy, since it will ensure best safety, in the expense of slow writing outside a transaction.

The best performance will be achieved by combining the two previous options, as such:

```pascal
Client := TSQLRestClientDB.Create(Model,nil,MainDBFileName,TSQLRestServerDB,false,'');
Client.Server.DB.LockingMode := lmExclusive;
Client.Server.DB.Synchronous := smOff;
```

Or, for external tables:

```pascal
Props := TSQLDBSQLite3ConnectionProperties.Create(DBFileName,'','','');
VirtualTableExternalRegister(Model,TSQLRecordSample,Props,'SampleRecord');
Client := TSQLRestClientDB.Create(Model,nil,MainDBFileName,TSQLRestServerDB,false,'');
TSQLDBSQLite3Connection(Props.MainConnection).Synchronous := smOff;
```

If you can afford loosing some data in very rare border case, or if you are sure your hardware configuration is safe (e.g. if the server is connected to a power inverter and has RAID disks) and that you have backups at hand, setting Synchronous := smOff will help your application scale for writing.
Setting LockingMode := lmExclusive will benefit of both writing and reading speed. Consider using an external and dedicated database (like Firebird, Oracle, PostgreSQL, MySQL, DB2, Informix or MS SQL) if your security expectations are very high, and if the default safe but slow setting is not enough for you.

### 7.2.8. Database backup

In all cases, do not forget to perform backups of your SQLite3 database as often as possible (at least several times a day). Adding a backup feature on the server side is as simple as running:

```pascal
Server.DB.BackupBackground('backup.db3',1024,10,nil);
```

The above line will perform a background live backup of the main SQLite3 database, by steps of 1024 pages (i.e. it will process 1 MB per step, since default page size is 1024 bytes), performing a little sleep of 10 milliseconds between each 1 MB copy step, allowing main CRUD / ORM operations to continue uninterrupted during the backup.

You can even specify an OnProgress: TSQLDatabaseBackupEvent callback event, to monitor the backup process.

Note that TSQLRestServerDB.Backup or TSQLRestServerDB.BackupGZ methods are not recommended any more on a running mORMot database, due to some potential issues with virtual tables, especially on the Win64 platform. You should definitively use TSQLDatabase.BackupBackground() instead.

The same backup process can be used e.g. to save an in-memory SQLite3 database into a SQLite3 file, as such:

```pascal
if aInMemoryDB.BackupBackground('backup.db3',-1,0,nil) then
  aInMemoryDB.BackupBackgroundWaitUntilFinished;
```

Above code will save the aInMemoryDB database into the 'backup.db3' file.
7.3. Virtual Tables magic

The SQLite3 engine has the unique ability to create Virtual Tables from code. From the perspective of an SQL statement, the virtual table object looks like any other table or view. But behind the scenes, queries from and updates to a virtual table invoke callback methods on the virtual table object instead of reading and writing to the database file.

The virtual table mechanism allows an application to publish interfaces that are accessible from SQL statements as if they were tables. SQL statements can in general do anything to a virtual table that they can do to a real table, with the following exceptions:

- One cannot create a trigger on a virtual table.
- One cannot create additional indices on a virtual table. (Virtual tables can have indices but that must be built into the virtual table implementation. Indices cannot be added separately using CREATE INDEX statements.)
- One cannot run ALTER TABLE ... ADD COLUMN commands against a virtual table.
- Particular virtual table implementations might impose additional constraints. For example, some virtual implementations might provide read-only tables. Or some virtual table implementations might allow INSERT or DELETE but not UPDATE. Or some virtual table implementations might limit the kinds of UPDATEs that can be made.

Example of virtual tables, already included in the SQLite3 engine, are FTS or RTREE tables.

Our framework introduces new types of custom virtual table. You'll find classes like TSQLVirtualTableJSON or TSQLVirtualTableBinary which handle in-memory data structures. Or it might represent a view of data on disk that is not in the SQLite3 format (e.g. TSQLVirtualTableLog). It can be used to access any external database, just as if they were native SQLite3 tables - see below (page 239). Or the application might compute the content of the virtual table on demand.

Thanks to the generic implementation of Virtual Table in SQLite3, you can use such tables in your SQL statement, and even safely execute a SELECT statement with JOIN or custom functions, mixing normal SQLite3 tables and any other Virtual Table. From the ORM point of view, virtual tables are just tables, i.e. they inherit from TSQLRecordVirtual, which inherits from the common base TSQLRecord class.

7.3.1. Virtual Table module classes

A dedicated mechanism has been added to the framework, beginning with revision 1.13, in order to easily add such virtual tables with pure Delphi code.

In order to implement a new Virtual Table type, you'll have to define a so called Module to handle the fields and data access and an associated Cursor for the SELECT statements. This is implemented by the two TSQLVirtualTable and TSQLVirtualTableCursor classes as defined in the mORMot.pas unit.

For instance, here are the default Virtual Table classes deriving from those classes:
Virtual Tables classes hierarchy

TSQLVirtualTableJSON, TSQLVirtualTableBinary and TSQLVirtualTableCursorJSON classes will implement a Virtual Table using a TSQLRestStorageInMemory instance to handle fast in-memory static databases. Disk storage will be encoded either as UTF-8 JSON (for the TSQLVirtualTableJSON class, i.e. the 'JSON' module), or in a proprietary SynLZ compressed format (for the TSQLVirtualTableBinary class, i.e. the 'Binary' module). File extension on disk will be simply .json for the 'JSON' module, and .data for the 'Binary' module. Just to mention the size on disk difference, the 502 KB People.json content (as created by included regression tests) is stored into a 92 KB People.data file, in our proprietary optimized format.

Note that the virtual table module name is retrieved from the class name. For instance, the TSQLVirtualTableJSON class will have its module named as 'JSON' in the SQL code.

To handle external databases, two dedicated classes, named TSQLVirtualTableExternal and TSQLVirtualTableCursorExternal will be defined in a similar manner - see External Databases classes hierarchy below (page 273).

As you probably have already stated, all those Virtual Table mechanism is implemented in mORMot.pas. Therefore, it is independent from the SQLite3 engine, even if, to my knowledge, there is no other SQL database engine around able to implement this pretty nice feature.

7.3.2. Defining a Virtual Table module

Here is how the TSQLVirtualTableLog class type is defined, which will implement a Virtual Table module named "Log". Adding a new module is just made by overriding some Delphi methods:

```delphi
TSQLVirtualTableLog = class(TSQLVirtualTable)
protected
  fLogFile: TSynLogFile;
public
  class procedure GetTableModuleProperties(var aProperties: TVirtualTableModuleProperties); override;
constructor Create(aModule: TSQLVirtualTableModule; const aTableName: RawUTF8; FieldCount: integer; Fields: PPUTF8CharArray); override;
destructor Destroy; override;
end;
```

This module will allow direct Read-Only access to a .log file content, which file name will be specified by the corresponding SQL table name.

The following method will define the properties of this Virtual Table Module:

```delphi
class procedure TSQLVirtualTableLog.GetTableModuleProperties( var aProperties: TVirtualTableModuleProperties);
begin
  aProperties.Features := [vtWhereIDPrepared];
  aProperties.CursorClass := TSQLVirtualTableCursorLog;
  aProperties.RecordClass := TSQLRecordLogFile;
end;
```

The supplied feature set defines a read-only module (since vtWrite is not selected), and vtWhereIDPrepared indicates that any RowID=? SQL statement will be handled as such in the cursor class (we will use the log row as ID number, start counting at 1, so we can speed up RowID=? WHERE clause easily). The associated cursor class is returned. And a TSQLRecord class is specified, to define the handled fields - its published properties definition will be used by the inherited Structure method to specify to the SQLite3 engine which kind of fields are expected in the SQL statements:

```delphi
TSQLRecordLogFile = class(TSQLRecordVirtualTableAutoID)
protected
```

...
You could have overridden the Structure method in order to provide the CREATE TABLE SQL statement expected. But using Delphi class RTTI allows the construction of this SQL statement with the appropriate column type and collation, common to what the rest of the ORM will expect.

Of course, this RecordClass property is not mandatory. For instance, the TSQLVirtualTableJSON.GetTableModuleProperties method won't return any associated TSQLRecordClass, since it will depend on the table it is implementing, i.e. the running TSQLRestStorageInMemory instance. Instead, the Structure method is overridden, and will return the corresponding field layout of each associated table.

Here is how the Prepare method is implemented, and will handle the vtWhereIDPrepared feature:

```delphi
function TSQLVirtualTable.Prepare(var Prepared: TSQLVirtualTablePrepared): boolean;
begin
  result := Self<>nil;
  if result then
    if (vtWhereIDPrepared in fModule.Features) and
      Prepared.IsWhereIDEquals(true) then
      with Prepared.Where[0] do
      begin
        // check ID=?
        Value.VType := varAny;
        // mark TSQLVirtualTableCursorJSON expects it
        OmitCheck := true;
        Prepared.EstimatedCost := 1;
      end
    else
      Prepared.EstimatedCost := 1E10; // generic high cost
  end;
end;
```

Then here is how each 'log' virtual table module instance is created:

```delphi
constructor TSQLVirtualTableLog.Create(aModule: TSQLVirtualTableModule;
var aTableName: RawUTF8; FieldCount: integer; Fields: PPUTF8CharArray);
var
  aFileName: TFileName;
begin
  inherited;
  if (FieldCount=1) then
    aFileName := UTF8ToString(Fields[0]) else
    aFileName := aModule.FileName(aTableName);
  fLogFile := TSynLogFile.Create(aFileName);
end;
```

It only associates a TSynLogFile instance according to the supplied file name (our SQL CREATE VIRTUAL TABLE statement only expects one parameter, which is the .log file name on disk - if this file name is not specified, it will use the SQL table name instead).

The TSQLVirtualTableLog.Destroy destructor will free this fLogFile instance:

```delphi
destructor TSQLVirtualTableLog.Destroy;
begin
  FreeAndNil(fLogFile);
  inherited;
end;
```

Then the corresponding cursor is defined as such:

```delphi
TSQLVirtualTableCursorLog = class(TSQLVirtualTableCursorIndex)
```
Since this class inherits from TSQLVirtualTableCursorIndex, it will have the generic fCurrent / fMax protected fields, and will have the HasData, Next and Search methods using those properties to handle navigation throughout the cursor.

The overridden Search method consists only in:

```pascal
function TSQLVirtualTableCursorLog.Search(const Prepared: TSQLVirtualTablePrepared): boolean;
begin
  result := inherited Search(Prepared); // mark EOF by default
  if result then begin
    fMax := TSQLVirtualTableLog(Table).fLogFile.Count - 1;
    if Prepared.IsWhereIDEquals(false) then begin
      fCurrent := Prepared.Where[0].Value.VInt64 - 1; // ID=? -> index := ID-1
      if cardinal(fCurrent)<=cardinal(fMax) then
        fMax := fCurrent else // found one
        fMax := fCurrent - 1; // out of range ID
    end;
  end;
end;
```

The only purpose of this method is to handle RowID=? statement SELECT WHERE clause, returning fCurrent=fMax=ID-1 for any valid ID, or fMax<fCurrent, i.e. no result if the ID is out of range. In fact, the Search method of the cursor class must handle all cases which has been notified as handled during the call to the Prepare method. In our case, since we have set the vtWhereIDPrepared feature and the Prepare method identified it in the request and set the OmitCheck flag, our Search method MUST handle the RowID=? case.

If the WHERE clause is not RowID=? (i.e. if Prepared.IsWhereIDEquals returns false), it will return fCurrent=0 and fMax=fLogFile.Count-1, i.e. it will let the SQLite3 engine loop through all rows searching for the data.

Each column value is retrieved by this method:

```pascal
function TSQLVirtualTableCursorLog.Column(aColumn: integer; var aResult: TVarData): boolean;
begin
  LogFile := TSQLVirtualTableLog(Table).fLogFile;
  if LogFile=nil then exit;
  case aColumn of
    0: SetColumn(aResult,LogFile.EventDateTime(fCurrent)); // ID = index + 1
    1: SetColumn(aResult,ord(LogFile.EventLevel[fCurrent]));
    2: SetColumn(aResult,LogFile.LinePointers[fCurrent],LogFile.LineSize(fCurrent));
    else exit;
  end;
  result := true;
end;
```

As stated by the documentation of the TSQLVirtualTableCursor class, -1 is the column index for the RowID, and then will follow the columns as defined in the text returned by the Structure method (in our case, the DateTime, Level, Content fields of TSQLRecordLogFile).

The SetColumn overloaded methods can be used to set the appropriate result to the aResult.
variable. For UTF-8 text, it will use a temporary in-memory space, to ensure that the text memory will be still available at least until the next Column method call.

7.3.3. Using a Virtual Table module

From the low-level SQLite3 point of view, here is how this "Log" virtual table module can be used, directly from the SQLite3 engine.

First we will register this module to a DB connection (this method is to be used only in case of such low-level access - in our ORM you should never call this method, but TSQLModel. VirtualTableRegister instead, cf. next paragraph):

```
RegisterVirtualTableModule(TSQLVirtualTableLog,Demo);
```

Then we can execute the following SQL statement to create the virtual table for the Demo database connection:

```
Demo.Execute('CREATE VIRTUAL TABLE test USING log(temptest.log);');
```

This will create the virtual table. Since all fields are already known by the TSQLVirtualTableLog class, it is not necessary to specify the fields at this level. We only specify the log file name, which will be retrieved by TSQLVirtualTableLog. Create constructor.

```
Demo.Execute('select count(*) from test',Res);
Check(Res=1);
s := Demo.ExecuteJSON('select * from test');
s2 := Demo.ExecuteJSON('select * from test where rowid=1');
s3 := Demo.ExecuteJSON('select * from test where level=3');
```

You can note that there is no difference with a normal SQLite3 table, from the SQL point of view. In fact, the full power of the SQL language as implemented by SQLite3 - see http://sqlite.org/lang.html - can be used with any kind of data, if you define the appropriate methods of a corresponding Virtual Table module.

7.3.4. Virtual Table, ORM and TSQLRecord

The framework ORM is able to use Virtual Table modules, just by defining some TSQLRecord, inheriting from some TSQLRecordVirtual dedicated classes:

```
Custom Virtual Tables records classes hierarchy
```

TSQLRecordVirtualTableAutoID children can be defined for Virtual Table implemented in Delphi, with a new ID generated automatically at INSERT.
**TSQLRecordVirtualTableForcedID** children can be defined for Virtual Table implemented in *Delphi*, with an ID value forced at **INSERT** (in a similar manner than for TSQLRecordRTree or TSQLRecordFTS3/FTS4/FTS5).

TSQLRecordLogFile was defined to map the column name as retrieved by the TSQLVirtualTableLog ('log') module, and should not to be used for any other purpose.

The Virtual Table module associated from such classes is retrieved from an association made to the server TSQLModel. In a Client-Server application, the association is not needed (nor to be used, since it may increase code size) on the Client side. But on the server side, the TSQLModel. VirtualTableRegister method must be called to associate a TSQLVirtualTableClass (i.e. a Virtual Table module implementation) to a TSQLRecordVirtualClass (i.e. its ORM representation).

For instance, the following code will register two TSQLRecord classes, the first using the 'JSON' virtual table module, the second using the 'Binary' module:

```delphi
Model.VirtualTableRegister(TSQLRecordDali1,TSQLVirtualTableJSON);
Model.VirtualTableRegister(TSQLRecordDali2,TSQLVirtualTableBinary);
```

This registration should be done on the Server side only, **before calling TSQLRestServer.Create** (or TSQLRestClientDB.Create, for a standalone application). Otherwise, an exception is raised at virtual table creation.

### 7.3.5. In-Memory "static" process

We have seen that the TSQLVirtualTableJSON, TSQLVirtualTableBinary and TSQLVirtualTableCursorJSON classes implement a Virtual Table module using a TSQLRestStorageInMemory instance to handle fast static in-memory database.

Why use such a database type, when you can create a *SQLite3* in-memory table, using the :memory: file name? That is the question...

- *SQLite3* in-memory tables are not persistent, whereas our JSON or Binary virtual table modules can be written on disk on purpose, if the aServer.StaticVirtualTable[aClass].CommitShouldNotUpdateFile property is set to true - in this case, file writing should be made by calling explicitly the aServer.StaticVirtualTable[aClass].UpdateToFile method;
- *SQLite3* in-memory tables will need two database connections, or call to the ATTACH DATABASE SQL statement - both of them are not handled natively by our Client-Server framework;
- *SQLite3* in-memory tables are only accessed via SQL statements, whereas TSQLRestStorageInMemory tables can have faster direct access for most common RESTful commands (GET / POST / PUT / DELETE individual rows) - this could make a difference in server CPU load, especially with the Batch feature of the framework;
- On the server side, it could be very convenient to have a direct list of in-memory TSQLRecord instances to work with in pure *Delphi* code; this is exactly what TSQLRestStorageInMemory allows, and definitively makes sense for an ORM framework;
- On the client or server side, you could create calculated fields easily with TSQLRestStorageInMemory dedicated "getter" methods written in *Delphi*, whereas *SQLite3* in-memory tables will need additional SQL coding;
- *SQLite3* tables are stored in the main database file - in some cases, it could be much convenient to provide some additional table content in some separated database file (for a round robin table, a configuration table written in JSON, some content to be shared among users...); this is made possible using our JSON or Binary virtual table modules (but, to be honest, the ATTACH DATABASE statement could provide a similar feature);
The TSQLRestStorageInMemory class can be used stand-alone, i.e. without the SQLite3 engine so it could be used to produce small efficient server software - see the "SQLite3\Samples\01 - In Memory ORM" folder.

### 7.3.5.1. In-Memory tables

A first way of using static tables, independently from the SQLite3 engine, is to call the TSQLRestServer. StaticDataCreate method.

This method is only to be called server-side, of course. For the Client, there is no difference between a regular and a static table.

The in-memory TSQLRestStorageInMemory instance handling the storage can be accessed later via the StaticDataServer[] property array of TSQLRestServer.

As we just stated, this primitive but efficient database engine can be used without need of the SQLite3 database engine to be linked to the executable, saving some KB of code if necessary. It will be enough to handle most basic RESTful requests.

### 7.3.5.2. In-Memory virtual tables

A more advanced and powerful way of using static tables is to define some classes inheriting from TSQLRecordVirtualTableAutoID, and associate them with some TSQLVirtualTable classes. The TSQLRecordVirtualTableAutoID parent class will specify that associated virtual table modules will behave like normal SQLite3 tables, so will have their RowID property computed at INSERT).

For instance, the supplied regression tests define such two tables with three columns, named FirstName, YearOfBirth and YearOfDeath, after the published properties definition:

```pascal
TSQLRecordDal11 = class(TSQLRecordVirtualTableAutoID)
private
  fYearOfBirth: integer;
  fFirstName: RawUTF8;
  fYearOfDeath: word;
published
  property FirstName: RawUTF8 read fFirstName write fFirstName;
  property YearOfBirth: integer read fYearOfBirth write fYearOfBirth;
  property YearOfDeath: word read fYearOfDeath write fYearOfDeath;
end;
TSQLRecordDal112 = class(TSQLRecordDal11);
```

Both class types are then added to the TSQLModel instance of the application, common to both Client and Server side:

```pascal
ModelC := TSQLModel.Create(
  [TSQLRecordPeople, (...)
  TSQLRecordDal11,TSQLRecordDal112], 'root');
```

Then, on the Server side, the corresponding Virtual Table modules are associated with those both classes:

```pascal
ModelC.VirtualTableRegister(TSQLRecordDal11,TSQLVirtualTableJSON);
ModelC.VirtualTableRegister(TSQLRecordDal112,TSQLVirtualTableBinary);
```

Thanks to the VirtualTableRegister calls, on the server side, the 'JSON' and 'Binary' Virtual Table modules will be launched automatically when the SQLite3 DB connection will be initialized:

```pascal
Client := TSQLRestClientDB.Create(ModelC,nil,Demo,TSQLRestServerTest);
```

This TSQLRestClientDB has in fact a TSQLRestServerDB instance running, which will be used for all Database access, including Virtual Table process.
Two files will be created on disk, named 'Dali1.json' and 'Dali2.data'. As stated above, the JSON version will be much bigger, but also more easy to handle from outside the application.

From the code point of view, there is no difference in our ORM with handling those virtual tables, compared to regular TSQLRecord tables. For instance, here is some code extracted from the supplied regression tests:

```pascal
if aClient.TransactionBegin(TSQLRecordDali1) then
  try
    // add some items to the file
    V2.FillPrepare(aClient, 'LastName:="Dali":');
    n := 0;
    while V2.FillOne do begin
      VD.FirstName := V2.FirstName;
      VD.YearOfBirth := V2.YearOfBirth;
      VD.YearOfDeath := V2.YearOfDeath;
      inc(n);
      Check(aClient.Add(VD,true)=n,Msg);
    end;
    // update some items in the file
    for i := 1 to n do begin
      Check(aClient.Retrieve(i,VD),Msg);
      Check(VD.ID=i);
      Check(IdemPChar(pointer(VD.FirstName), 'SALVADOR'));
      Check(VD.YearOfBirth=1904);
      Check(VD.YearOfDeath=1989);
      VD.YearOfBirth := VD.YearOfBirth+i;
      VD.YearOfDeath := VD.YearOfDeath+i;
      Check(aClient.Update(VD),Msg);
    end;
    // check SQL requests
    for i := 1 to n do begin
      Check(aClient.Retrieve(i,VD),Msg);
      Check(VD.YearOfBirth=1904+i);
      Check(VD.YearOfDeath=1989+i);
    end;
    Check(aClient.TableRowCount(TSQLRecordDali1)=1001);
  aClient.Commit;
  except
    aClient.RollBack;
  end;
```

A Commit is needed from the Client side to write anything on disk. From the Server side, in order to create disk content, you'll have to explicitly call such code on purpose:

As we already noticed, data will be written by default on disk with our TSQLRestStorageInMemory-based virtual tables. In fact, the Commit method in the above code will call TSQLRestStorageInMemory.UpdateFile.

Please note that the SQLite3 engine will handle any Virtual Table just like regular SQLite3 tables, concerning the atomicity of the data. That is, if no explicit transaction is defined (via TransactionBegin / Commit methods), such a transaction will be performed for every database modification (i.e. all CRUD operations, as INSERT / UPDATE / DELETE). The TSQLRestStorageInMemory. UpdateToFile method is not immediate, because it will write all table data each time on disk. It is therefore mandatory, for performance reasons, to nest multiple modification to a Virtual Table with such a transaction, for better performance. And in all cases, it is the standard way of using the ORM. If for some reason, you later change your mind and e.g. move your table from the TSQLVirtualTableJSON / TSQLVirtualTableBinary engine to the default SQLite3 engine, your code could remain untouched.

It is possible to force the In-Memory virtual table data to stay in memory, and the COMMIT statement
to write nothing on disk, using the following property:

```plaintext
Server.StaticVirtualTable[TSQLRecordDali1].CommitShouldNotUpdateFile := true;
```

In order to create disk content, you'll then have to explicitly call the corresponding method on purpose:

```plaintext
Server.StaticVirtualTable[TSQLRecordDali1].UpdateToFile;
```

Since StaticVirtualTable property is only available on the Server side, you are the one to blame if your client updates the table data and this update never reaches the disk!

### 7.3.5.3. In-Memory and ACID

For data stored in memory, the TSQLRestStorageInMemory table is ACID. It means that concurrent access will be consistent and work safely, as expected.

On disk, this kind of table is ACID only when its content is written to the file. I mean, the whole file which will be written in an ACID way. The file will always be consistent.

The exact process of these in-memory tables is that each time you write some new data to a TSQLRestStorageInMemory table:
- It will be ACID in memory (i.e. work safely in concurrent mode);
- Individual writes (INSERT/UPDATE/DELETE) won't automatically be written to file;
- COMMIT will by default write the whole table to file (either as JSON or compressed binary);
- ROLLBACK process won't do anything, so won't be ACID - but since your code may later use a real RDBMS, it is a good habit to always write the command, like in the sample code above, as except aClient.Rollback.

When you write the data to file, the whole file is rewritten: it seems not feasible to write the data to disk at every write - in this case, SQLite3 in exclusive mode will be faster, since it will write only the new data, not the whole table content.

This may sound like a limitation, but on our eyes, it could be seen more like a feature. For a particular table, we do not need nor want to have a whole RDBMS/SQL engine, just direct and fast access to a TObjectList. The feature is to integrate it with our REST engine, and still be able to store your data in a regular database later (SQLite3 or external), if it appears that TSQLRestStorageInMemory storage is too limited for your process.

### 7.3.6. Redirect to an external TSQLRest

Sometimes, having all database process hosted in a single process may not be enough. You can use the TSQLRestServer.RemoteDataCreate() method to instantiate a TSQLRestStorageRemote class which will redirect all ORM operation to a specified TSQLRest instance, may be remote (via TSQLRestClientHttp) or in-process (TSQLRestServer). REST redirection may be enough in simple use cases, when full Master/slave replication (page 180) could be oversized.

For instance, in TTestExternalDatabase regression tests, you will find the following code:

```plaintext
aExternalClient := TSQLRestClientDB.Create(fExternalModel, nil, 'testExternal.db3', TSQLRestServerDB);
historyDB := TSQLRestServerDB.Create(
  TSQLModel.Create([TSQLRecordMyHistory], 'history'),
  'history.db3', false);
historyDB.Model.Owner := historyDB;
historyDB.DB.Synchronous := smOff;
historyDB.DB.LockingMode := lmExclusive;
```
historyDB.CreateMissingTables;
    'history.db3',false);
aExternalClient.Server.RemoteDataCreate(TSQLRecordMyHistory,historyDB);
aExternalClient.Server.DB.Synchronous := smOff;
aExternalClient.Server.DB.LockingMode := lmExclusive;
aExternalClient.Server.CreateMissingTables;
...

It will create two SQLite3 databases, one main "testExternal.db3", and a separated "history.db3" database. Both will use synch off and lock exclusive access mode - see ACID and speed (page 222) just above.

In the "history.db3" file, there will be the MyHistory table, whereas in testExternal.db3", there won't be any MyHistory table. All TSQLRecordMyHistory CRUD process will be transparently redirected to historyDB.

Then any ORM access from the main aExternalClient to the TSQLRecordMyHistory table via will be redirected, via an hidden TSQLRestStorageRemote instance, to historyDB. There won't be any noticeable performance penalty - on the contrary a separated database will be much better.

An alternative may have been to use the ATTACH TABLE statement at SQLite3 level, but it will have been only locally, and you will not be able to switch to another database engine. Whereas the RemoteDataCreate() method is generic, and will work with external databases - see below (page 239), even NoSQL databases - see below (page 287), or remote mORMot servers, accessible via a TSQLRestClientHTTP instance. The only prerequisite is that all TSQLRecord classes in the main model do exist in the redirected database model.

Note that the redirected TSQLRest instance can have its own model, its own authentication and authorization scheme, its own caching policy. It may be of great interest when tuning your application.

Be aware that if you use TRecordReference published fields, the model should better be shared among the local and redirected TSQLRest instances, or at least the TSQLRecord classes should have the same order - otherwise the TRecordReference values will point to the wrong table, depending on the side the query is run.

See General mORMot architecture - Client Server implementation (page 82) and Client-Server implementation - Server side for some explanation about how this redirection feature interacts with other abilities of the framework.

One practical application of this redirection pattern may be with a typical corporate business.
There may be a main mORMot server, at corporation headquarters, then local mORMot servers at each branch office, hosting applications for end users on the local network:
Each branch office may have its own TSQLRecord dedicated table, with all its data. Some other tables will be shared among local offices, like global configuration.

Creating a dedicated table can be done in Delphi code by creating your own class type:

```delphi
type
  TSQLRecordCustomerAbstract = class // never declared in Model
  .... // here the fields used for Customer business
end;

TSQLRecordCustomerA = class(TSQLRecordCustomerAbstract); // for office A
TSQLRecordCustomerB = class(TSQLRecordCustomerAbstract); // for office B

TSQLRecordCustomerClass = class of TSQLRecordCustomerAbstract;
```

Here, TSQLRecordCustomerA may be part only of the Office A server's TSQLModel, and TSQLRecordCustomerB only of the Office B server's TSQLModel. It will increase security, and, in the main headquarters server, both TSQLRecordCustomerA and TSQLRecordCustomerB classes will be part of the TSQLModel, and dedicated interface-based services will be able to publish some high-level data and statistics about all stored tables.

Then you can use a TSQLRecordCustomerClass variable in your client code, which will contain either TSQLRecordCustomerA or TSQLRecordCustomerB, depending on the place it runs on, and the server it is connected to.

On the main server, each office will have its own storage table in the (external) database, named CustomerA or CustomerB.

You will benefit of the caching abilities - see below (page 359) - of each TSQLRest instance. You may have some cache tuned at a local site, whereas the cache in the main database will remain less aggressive, but safer.

Furthermore, even on a single-siter server, a TSQLRecordHistory table, or more generally any aggregation data may benefit to be hosted locally or on cheap storage, whereas the main database will stay on SSD or SAS. Thanks to this redirection feature, you can tune your hosting as expected.

Finally, if your purpose is to redirect all tables of a given TSQLRestServer to another remote TSQLRestServer (for security or hosting purpose), you may consider using TSQLRestServerRemoteDB instead. This class will redirect all tables to one external instance.

Note that both TSQLRestStorageRemote and TSQLRestServerRemoteDB classes do not support yet the Virtual Tables mechanism of SQLite3. So if you use those features, you may not be able to run...
JOINed queries from the redirected instance: in fact, the main SQLite3 engine will complain about a missing MyHistory table in "testExternal1.db3". We will eventually define the needed TSQLVirtualTableRemote and TSQLVirtualTableCursorRemote classes to implement this feature.

Sadly, this redirection pattern won't work if the connection is lost. The main office server needs to be always accessible so that the local offices continue to work. You may consider using Master/slave replication (page 180) to allow the local offices to work with their own local copy of the master data. Replication sounds in fact preferred than simple redirection, especially in terms of network and resource use, in some cases.

### 7.3.7. Virtual Tables to access external databases

As will be stated below (page 239), some external databases may be accessed by our ORM.

The Virtual Table feature of SQLite3 will allow those remote tables to be accessed just like "native" SQLite3 tables - in fact, you may be able e.g. to write a valid SQL query with a JOIN between SQLite3 tables, MS SQL Server, MySQL, FireBird, PostgreSQL, MySQL, DB2, Informix and Oracle databases, even with multiple connections and several remote servers. Think as an ORM-based Business Intelligence from any database source. Added to our code-based reporting engine (able to generate pdf), it could be a very powerful way of consolidating any kind of data.

In order to define such external tables, you define your regular TSQLRecord classes as usual, then a call to the VirtualTableExternalRegister() or VirtualTableExternalMap() functions will define this class to be managed as a virtual table, from an external database engine. Using a dedicated external database server may allow better response time or additional features (like data sharing with other applications or languages). Server-side may omit a call to VirtualTableExternalRegister() if the need of an internal database is expected: it will allow custom database configuration at runtime, depending on the customer’s expectations (or license).

### 7.3.8. Virtual tables from the client side

For external databases - see below (page 239) - the SQL conversion will be done on the fly in a more advanced way, so you should be able to work with such virtual tables from the client side without any specific model notification. In this case, you can safely define your tables as TSQLValue1 = class(TSQLRecord), with no further code on client side.

When working with static (in-memory / TObjectList) storage, if you expect all ORM features to work remotely, you need to notify the Client-side model that a table is implemented as virtual. Otherwise you may encounter some SQL errors when executing requests, like "no such column: ID".

For instance, imagine you defined two in-memory JSON virtual tables on Server side:

```pascal
type
TSQLServer = class(TSQLRestServerDB)
private
  FHttpServer: TSQLHTTPServer;
public
  constructor Create;
  destructor Destroy; override;
end;

constructor TSQLServer.Create;
var
  aModel: TSQLModel;
begin
  aModel := CreateModel;
  aModel.VirtualTableRegister(TSQLValue1, TSQLVirtualTableJSON);
```
You will need to specify also on the client side that those TSQLValue1 and TSQLValue2 tables are virtual.

You have several possibilities:
- Inherit each table not from TSQLRecord, but from TSQLRecordVirtualTableAutoID, as was stated above as standard procedure for virtual tables - see In-Memory virtual tables (page 231);
- If your tables are defined as TSQLRecord, ensure that the Client side set the table property of its own model to rCustomAutoID;
- If your tables are defined as TSQLRecord, ensure that both Client and Server set the table property of its own model to rCustomAutoID.

First option could be done as such:

```pascal
type
  TSQLValue1 = class(TSQLRecordVirtualTableAutoID)
    (....)
  TSQLValue2 = class(TSQLRecordVirtualTableAutoID)
    (....)
```

Or, in case the table is defined as TSQLValue1 = class(TSQLRecord), the client model could be updated as such:

```pascal
type
  TSQLClient = class(TSQLHttpClient)
  public
    constructor Create;
  end;

constructor TSQLClient.Create;
var
  aModel: TSQLModel;
begin
  aModel:= CreateModel;
  aModel.Owner := self; // model will be released within TSQLServer instance
  aModel.Create(aModel, ChangeFileExt(ParamStr(0), '.db'), True);
  Self.CreateMissingTables(0);
  FHttpServer:= TSQLHttpServer.Create('8080', Self);
end;

destructor TSQLServer.Destroy;
begin
  FHttpServer.Free;
  inherited;
end;
```

Or, in case the table is defined as TSQLValue1 = class(TSQLRecord), perhaps the easiest way of doing it, is to set the property when creating the shared model:

```pascal
function CreateModel: TSQLModel;
begin
  result:= TSQLModel.Create([TSQLAuthGroup, TSQLAuthUser, TSQLValue1, TSQLValue2]);
  result.Props[TSQLValue1].Kind := rCustomAutoID;
  result.Props[TSQLValue2].Kind := rCustomAutoID;
  result.Owner := self; // model will be released within TSQLServer instance
  inherited Create('127.0.0.1', '8080', aModel);
  SetUser('Admin', 'synopse');
end;
```

The easiest is definitively to let your static in-memory tables inherit from TSQLRecordVirtualTableAutoID. Just use the framework by the book - see In-Memory virtual tables
(page 231).

Once again, this restriction does not apply to below (page 239).
8. External SQL database access

Our ORM RESTful framework is able to access most available database engines, via a set of generic units and classes. Both SQL and NoSQL engines could be accessed - quite a unique feature in the ORM landscape (in Delphi, of course, but also in Java or C# environments).

Remember the diagram introducing **mORMot’s Database layer** (page 196):

![mORMot Persistence Layer Architecture](image_url)
The framework still relies on SQLite3 as its SQL core on the server, but a dedicated mechanism allows access to any remote database, and mixes those tables content with the native ORM tables of the framework. Thanks to the unique Virtual Tables magic (page 225) mechanism of SQLite3, those external tables may be accessed as native SQLite3 tables in our SQL statements, even for NoSQL engines.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Engines</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL</td>
<td>SQLite3, Oracle, NexusDB, MS SQL, Jet/MSAccess, FireBird, MySQL, PostgreSQL, IBM DB2, IBM Informix</td>
</tr>
<tr>
<td></td>
<td>See below (page 241)</td>
</tr>
<tr>
<td>NoSQL</td>
<td>MongoDB, TObjectList with JSON or binary disk persistence</td>
</tr>
<tr>
<td></td>
<td>See below (page 278) and In-Memory &quot;static&quot; process (page 230)</td>
</tr>
</tbody>
</table>

You can even mix databases, i.e. the same mORMot ORM could persist, at the same time, its data in several databases, some TSQLRecord as fast internal SQLite3 tables or as TObjectList, others in a PostgreSQL database (tied to an external reporting/SAP engine), and e.g. flat consolidated data in a MongoDB instance.
8.1. SynDB direct RDBMS access

External Relational Database Management System (RDBMS) can be accessed via our SynDB.pas units. Then, the framework ORM is able to access them via the mORMotDB.pas bridge unit. But you can use the SynDB.pas units directly, without any link to our ORM.

The current list of handled data access libraries is:

<table>
<thead>
<tr>
<th>Provider</th>
<th>SynDB Unit</th>
<th>RDBMS Engines</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite3</td>
<td>SynDBSQLite3.pas</td>
<td>direct SQLite3 access (as dll or linked to the exe)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See below (page 259)</td>
</tr>
<tr>
<td>Oracle</td>
<td>SynDBOracle.pas</td>
<td>direct Oracle access (via OCI)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See below (page 256)</td>
</tr>
<tr>
<td>OleDB</td>
<td>SynOleDb.pas</td>
<td>MS SQL, Jet/MSAccess or others</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See below (page 253)</td>
</tr>
<tr>
<td>ODBC</td>
<td>SynDBODBC.pas</td>
<td>MS SQL, FireBird, MySQL, PostgreSQL, IBM DB2, Informix or others</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See below (page 253)</td>
</tr>
<tr>
<td>ZeosLib</td>
<td>SynDBZeos.pas</td>
<td>MS SQL, SQLite3, FireBird, MySQL, PostgreSQL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See below (page 254)</td>
</tr>
<tr>
<td>DB.pas/T Dataset</td>
<td>SynDBDataset.pas</td>
<td>NexusDB and databases supported by DBExpress, FireDAC, AnyDac, UniDAC, BDE...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See below (page 259)</td>
</tr>
<tr>
<td>HTTP</td>
<td>SynDBRemote.pas</td>
<td>remote access to any SynDB database, over HTTP</td>
</tr>
</tbody>
</table>

This list is not closed, and may be completed in the near future. Any help is welcome here: it is not difficult to implement a new unit, following the patterns already existing. You may start from an existing driver (e.g. Zeos or Alcinoe libraries). Open Source contribution are always welcome!

Thanks to the design of our SynDB.pas classes, it was very easy (and convenient) to implement SQLite3 direct access. It is even used for our regression tests, in order to implement stand-alone unitary testing.

An Oracle dedicated direct access was added, because all available OleDB providers for Oracle (i.e. both Microsoft’s and Oracle’s) do have problems with handling BLOB, and we wanted our Clients to have a light-weight and as fast as possible access to this great database.

In fact, OleDB is a good candidate for database access with good performance, Unicode native, with a lot of available providers. Thanks to OleDB, we are already able to access to almost any existing database. The code overhead in the server executable will also be much less than with adding any other third-party Delphi library. And we will let Microsoft or the OleDB provider perform all the testing and debugging for each driver.

Since revision 1.17, direct access to the ODBC layer has been included to the framework database units. It has a wider range of free providers (including e.g. MySQL or FireBird), and is the official replacement for OleDB (next version of MS SQL Server will provide only ODBC providers, as far as Microsoft warned its customers).
Since revision 1.18, any ZeosLib / ZDBC driver can be used, with fast direct access to the underlying RDBMS client library. Since the ZDBC library does not rely on DB.pas, and by-passes the slow TDataSet component, its performance is very high. The ZDBC maintainers did a lot of optimizations, especially to work with mORMot, and this library is a first-class citizen to work with our framework.

Since the same 1.18 revision, DB.pas can be used with our SynDB.pas classes. Of course, using TDataSet as intermediate layer will be slower than the SynDB.pas direct access pattern. But it will allow you to re-use any existing (third-party) database connection driver, which could make sense in case of evolution of an existing application, or to use an unsupported database engine.

Last but not least, the SynDBRemote.pas unit allows you to create database applications that perform SQL operations on a remote SynDB HTTP server, instead of a database server. You can create connections just like any other SynDB database, but the transmission will take place over HTTP, with no need to install a database client with your application - see below (page 263).

The following connections are therefore possible:

This diagram is a bit difficult to follow at the latest level - but you got the general layered design, I guess. It will be split into smaller focused diagrams later.

**8.1.1. Direct access to any RDBMS engine**

The SynDB.pas units have the following features:

- Direct fast access via OleDB, ODBC, ZDBC, Oracle (OCI) or SQLite3 (statically linked or via external d11);
- Thin wrapper around any DB.pas / TDataSet based components (e.g. NexusDB, DBExpress, FireDAC, AnyDAC, UniDAC, BDE...);
- Generic abstract OOP layout, with a restricted set of data types, but able to work with any SQL-based database engine;
- Tested with MS SQL Server 2008/2012, Firebird 2.5.1, PostgreSQL 9.2/9.3, MySQL 5.6, IBM DB2 10.5, Oracle 11g, and the latest SQLite3 engine;
- Could access any local or remote Database, from any edition of Delphi (even Delphi 7 personal, the Turbo Explorer or Starter edition), just for free (in fact, it does not use the DB.pas standard unit and all its dependencies);
- Unicode, even with pre-Unicode version of Delphi (like Delphi 7 or 2007), since it uses internally UTF-8 encoding;
- Handle NULL or BLOB content for parameters and results, including stored procedures;
- Avoid most memory copy or unnecessary allocation: we tried to access the data directly from the retrieved data buffer, just as given from OleDB / ODBC or the low-level database client (e.g. OCI for Oracle, or the SQLite3 engine);
- Designed to achieve the best possible performance on 32-bit or 64-bit Windows: most time is spent...
in the database provider (OleDB, ODBC, OCI, SQLite3) - the code layer added to the database client
is very thin and optimized;
- Could be safely used in a multi-threaded application/server (with dedicated thread-safe methods, usable even if the database client is not officially multi-thread);
- Allow parameter bindings of prepared requests, with fast access to any parameter or column name (thanks to TDynArrayHashed);
- Column values accessible with most Delphi types, including Variant or generic string / WideString;
- Available ISQLDBRows interface - to avoid typing try...finally Query.Free end; and allow one-line SQL statement;
- Late-binding column access, via a custom variant type when accessing the result sets;
- Two kind of optimized TDataSet result sets: one read-write based on TClientDataSet, and a much faster read-only TSynSQLStatementDataSet
- Direct UTF-8 JSON content creation, with no temporary data copy nor allocation (this feature will be the most used in our JSON-based ORM server);
- High-level catalog / database layout abstract methods, able to retrieve the table and column properties (including indexes), for database reverse-engineering; provide also SQL statements to create a table or an index in a database-abstract manner; those features will be used directly by our ORM;
- Designed to be used with our ORM, but could be used stand-alone (a full Delphi 7 client executable is just about 200 KB), or even in any existing Delphi application, thanks to a TQuery-like wrapper;
- TQuery emulation class, for direct re-use with existing code, in replacement to DB.pas based code (including the deprecated BDE technology), with huge speed improvement for result sets (since we bypass the slow TDataSet component);
- Fast and safe remote access over HTTP to any SynDB engine, without the need to deploy the RDBMS client library with the application;
- Free SynDBExplorer tool provided, which is a small but efficient way of running queries in a simple User Interface, on all supported engines, and publish as server or consume as client SynDB remote access over HTTP - see below (page 263); it is also a good sample program of a stand-alone usage of those libraries.

8.1.2. Data types

Of course, our ORM does not need a whole feature set (do not expect to use this database classes with your VCL DB RAD components), but handles directly the basic SQL column types, as needed by our ORM (derived from SQLite's internal column types): NULL, Int64, Double, Currency, DateTime, RawUTF8 and BLOB.

They are defined as such in SynDB.pas:

```pascal
TSQLDBFieldType = (
  ftUnknown, ftNull, ftInt64, ftDouble, ftCurrency, ftDate, ftUTF8, ftBlob);
```

<table>
<thead>
<tr>
<th>TSQLDBFieldType</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>ftNull</td>
<td>Maps the SQL NULL value</td>
</tr>
<tr>
<td>ftInt64</td>
<td>Any integer value, with 64-bit resolution</td>
</tr>
<tr>
<td>ftDouble</td>
<td>Any floating-point value, with 64-bit (double) resolution</td>
</tr>
<tr>
<td>ftCurrency</td>
<td>Fixed financial type, with up to 4 fixed decimal digits (currency)</td>
</tr>
</tbody>
</table>
**ftDate**  
Date and time, mapping the Delphi TDateTime type

**ftUTF8**  
Unicode text, encoded as UTF-8, with or without size limit

**ftBlob**  
Binary content, stored as a RawByteString

Those types will map low-level database-level access types, not high-level Delphi types as TSQLFieldType defined in mORMot.pas, or the generic huge TFieldType as defined in the standard VCL DB.pas unit. In fact, it is more tied to the standard SQLite3 generic types, i.e. NULL, INTEGER, REAL, TEXT, BLOB (with the addition of a ftCurrency and ftDate type, for better support of most DB engines) see [http://www.sqlite.org/datatype3.html](http://www.sqlite.org/datatype3.html).

You can note that the only string type handled here uses UTF-8 encoding (implemented using our RawUTF8 type), for cross-Delphi true Unicode process. Code can access to the textual data via variant, string or widestring variables and parameters, but our units will use UTF-8 encoding internally - see *Unicode and UTF-8* (page 105). It will therefore interface directly with our ORM, which uses the same encoding. Of course, if the column was not defined as Unicode text in the database, any needed conversion to/from the corresponding charset will take place at the data provider level; but in your user code, you will have always access to the Unicode content.

BLOB columns or parameters are accessed as RawByteString variables, which may be mapped to a standard TStream via our TRawByteStringStream.

### 8.1.3 Database types

In addition to raw data access, the SynDB.pas unit handles some SQL-level generation, which will be used by our Object-Relational Mapping (page 130) kernel.

The following RDBMS database engines are defined as such in SynDB.pas:

```pascal
TSQLDBDefinition = (dUnknown, dDefault, dOracle, dMSSQL, dJet, 
                    dMySQL, dSQLite, dFirebird, dNexusDB, dPostgreSQL, dDB2, dInformix);
```

<table>
<thead>
<tr>
<th>TSQLDBDefinition</th>
<th>RDBMS tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>dUnknown</td>
<td>Any database, following the SQL-92 standard</td>
</tr>
<tr>
<td>dDefault</td>
<td>Oracle 11g</td>
</tr>
<tr>
<td>dOracle</td>
<td>MS SQL Server 2008/2012</td>
</tr>
<tr>
<td>dMSSQL</td>
<td>Jet/MSAccess (under Win32 only)</td>
</tr>
<tr>
<td>dMySQL</td>
<td>MySQL 5.6</td>
</tr>
<tr>
<td>dSQLite</td>
<td>SQLite3 3.7.11 and up (we supply the latest version for static linking)</td>
</tr>
<tr>
<td>dFirebird</td>
<td>Firebird 2.5.1</td>
</tr>
<tr>
<td>dNexusDB</td>
<td>NexusDB 3.11</td>
</tr>
<tr>
<td>dPostgreSQL</td>
<td>PostgreSQL 9.2/9.3</td>
</tr>
<tr>
<td>dDB2</td>
<td>IBM DB2 10.5</td>
</tr>
<tr>
<td>dInformix</td>
<td>IBM Informix 11.70</td>
</tr>
</tbody>
</table>
The above versions have been tested, but newer or older revisions may also work. Your feedback is welcome: we cannot achieve to test all possible combinations of databases and clients on our own!

The SynDB.pas unit is able to generate the SQL statements of those engines, for a CREATE TABLE / CREATE INDEX command, retrieve metadata (e.g. the tables and fields information), compute the right limit/offset syntax for a SELECT, compute multi-INSERT statements - see below (page 357), check the SQL keywords, define specific schema/owner naming conventions, process date and time values, handle errors and exceptions, or even create a database.

### 8.1.4. SynDB Units

Here are the units implementing the external database-agnostic features:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynDB.pas</td>
<td>abstract database direct access classes</td>
</tr>
<tr>
<td>SynOleDB.pas</td>
<td>OleDB direct access classes</td>
</tr>
<tr>
<td>SynDBODBC.pas</td>
<td>ODBC direct access classes</td>
</tr>
<tr>
<td>SynDBZeos.pas</td>
<td>ZDBC direct access classes</td>
</tr>
<tr>
<td>SynDBOracle.pas</td>
<td>Oracle DB direct access classes (via OCI)</td>
</tr>
<tr>
<td>SynDBSQLite3.pas</td>
<td>SQLite3 direct access classes</td>
</tr>
<tr>
<td>SynDBDataset.pas</td>
<td>TDataset (DB.pas) access classes</td>
</tr>
<tr>
<td>SynDBFireDAC.pas</td>
<td></td>
</tr>
<tr>
<td>SynDBUniDAC.pas</td>
<td></td>
</tr>
<tr>
<td>SynDBNexusDB.pas</td>
<td></td>
</tr>
<tr>
<td>SynDBBDE.pas</td>
<td></td>
</tr>
<tr>
<td>SynDBRemote.pas</td>
<td>remote access over HTTP</td>
</tr>
<tr>
<td>SynDBVCL</td>
<td>read/only TSynSQLStatementDataSet result sets</td>
</tr>
<tr>
<td>SynDBMidasVCL</td>
<td>read/write TClientDataSet result sets</td>
</tr>
</tbody>
</table>

It is worth noting that those units only depend on SynCommons.pas, therefore are independent of the ORM part of our framework (even the remote access). They may be used separately, accessing all those external databases with regular SQL code. Since all their classes inherit from abstract classes defined in SynDB.pas, switching from one database engine to another (even a remote HTTP access) is just a matter of changing one class type.

### 8.1.5. SynDB Classes

The data is accessed via three families of classes:
- **Connection properties**, which store the database high-level properties (like database implementation classes, server and database name, user name and password);
- **Connections**, which implements an actual connection to a remote database, according to the specified Connection properties - of course, there can be multiple connections for the same connection properties instance;
- **Statements**, which are individual SQL queries or requests, which may be multiple for one existing connection.
In practice, you define a `TSQLDBConnectionProperties` instance, then you derivate `TSQLDBConnection` and `TSQLDBStatement` instances using dedicated `NewConnection` / `ThreadSafeConnection` / `NewStatement` methods.

Here is the general class hierarchy, for all available remote *connection properties*:

### TSQLDBConnectionProperties classes hierarchy

Those classes are the root classes of the SynDB.pas units, by which most of your database process will be implemented. For instance, the mORMot framework ORM only needs a given `TSQLDBConnectionProperties` instance to access any external database.

Then the following *connection* classes are defined:

### TSQLDBSQLite3Connection classes hierarchy

Each connection may create a corresponding *statement* instance:
In the above hierarchy, TSQLDBDatasetStatementAbstract is used to allow the use of custom classes for parameter process, e.g. TADParams for FireDAC (which features Array DML).

Some dedicated Exception classes are also defined:

Check the TestOleDB.dpr sample program, located in SQLite3 folder, using our SynOleDB unit to connect to a local MS SQL Server 2008 R2 Express edition, which will write a file with the JSON representation of the Person.Address table of the sample database AdventureWorks2008R2.

8.1.6. ISQLDBRows interface

The easiest is to stay at the TSQLDBConnectionProperties level, using the Execute() methods of this instance, and access any returned data via an ISQLDBRows interface. It will automatically use a thread-safe connection to the database, in an abstracted way.

Typical use of SynDB.pas classes could be:
- Initialize a shared TSQLDBConnectionProperties instance;
- Execute statements directly from this instance’s Execute*() methods.

Defining a database connection is as easy as:

```delphi
var Props: TSQLDBConnectionProperties;
...
Props := TOleDBMSSQLConnectionProperties.Create('.\SQLEXPRESS', 'AdventureWorks2008R2', '', '');
try
  UseProps(Props);
finally
  Props.Free;
end;
```

Depending on the TSQLDBConnectionProperties sub-class, input parameters do vary. Please refer to the documentation of each Create() constructor to set all parameters as expected.
Then any sub-code is able to execute any SQL request, with optional bound parameters, as such:

```pascal
procedure UseProps(Props: TSQLDBConnectionProperties);
var I: ISQLDBRows;
begin
  I := Props.Execute('select * from Sales.Customer where AccountNumber like ?', ['AW000001%']);
  while I.Step do
    assert(Copy(I['AccountNumber'], 1, 8) = 'AW000001');
end;
```

In this procedure, no TSQLDBStatement is defined, and there is no need to add a `try ... finally` block.

In fact, the MyConnProps.Execute method returns a TSQLDBStatement instance as a ISQLDBRows, which methods can be used to loop for each result row, and retrieve individual column values. In the code above, I['FirstName'] will in fact call the I.Column[] default property, which will return the column value as a variant. You have other dedicated methods, like ColumnUTF8 or ColumnInt, able to retrieve directly the expected data.

Note that all bound parameters will appear within the SQL statement, when logged using our TSynLog classes - see below (page 641).

### 8.1.7. Using properly the ISQLDBRows interface

You may have noticed in the previous code sample, that we used a UseProps() sub-procedure. This was made on purpose.

We may have written our little test as such:

```pascal
var Props: TSQLDBConnectionProperties;
I: ISQLDBRows;
... Props := TOleDBMSSQLConnectionProperties.Create('.\SQLEXPRESS','AdventureWorks2008R2','','');
try
  I := Props.Execute('select * from Sales.Customer where AccountNumber like ?', ['AW000001%']);
  while I.Step do
    assert(Copy(I['AccountNumber'], 1, 8) = 'AW000001');
finally
  Props.Free;
end;
end;
```

In fact, you should **not** use this pattern. This code will lead to an unexpected *access violation* at runtime.

Behind the scene, as will be detailed below (page 385), the compiler is generating some hidden code to finalize the I: ISQLDBRows local variable, as such:

```pascal
... finally
  Props.Free;
end;
I := nil; // this is generated by the compiler, just before the final "end;"
end;
```

So ISQLDBRows is released after the Props instance, and an access violation occurs.

The correct way to write is either to use a sub-function (which will release the local ISQLDBRows when the function leaves), or explicitly release the interface variable:

```pascal
... finally
  Props.Free;
end;
I := nil; // this is generated by the compiler, just before the final "end;"
end;
```

while I.Step do
  assert(Copy(I['AccountNumber'], 1, 8) = 'AW000001');
finally
Of course, most of the time you will initialize your TSQLDBConnectionProperties globally for your process, then release it when it ends. Each request will take place in its own sub-method, so will be released before the main TSQLDBConnectionProperties instance if freed.

Last but not least, it is worth writing that you should not create a TSQLDBConnectionProperties instance each time you need to access the database, since you will probably lose most of the SynDB features, like a per-thread connection pool, or statement cache.

### 8.1.8. Late-binding

We implemented late-binding access of column values, via a custom variant time. It uses the internal mechanism used for Ole Automation, here to access column content as if column names were native object properties.

The resulting Delphi code to write is just clear and obvious:

```delphi
procedure UseProps(Props: TSQLDBConnectionProperties);
var
  Row: Variant;
begin
  with Props.Execute('select * from Sales.Customer where AccountNumber like ?',
    ['AW000001%'],@Row) do
    while Step do
      assert(Copy(Row.AccountNumber,1,8)='AW000001');
end;
```

Note that Props.Execute returns an ISQLDBRows interface, so the code above will initialize (or reuse an existing) thread-safe connection (OleDb uses a per-thread model), initialize a statement, execute it, access the rows via the Step method and the Row variant, retrieving the column value via a direct Row.AccountNumber statement.

The above code is perfectly safe, and all memory will be released with the reference count garbage-collector feature of the ISQLDBRows interface. You are not required to add any try..finally Free; end statements in your code.

This is the magic of late-binding in Delphi. Note that a similar feature is available for our SynBigTable unit.

In practice, this code is slower than using a standard property based access, like this:

```delphi
while Step do
  assert(Copy(ColumnUTF8('AccountNumber'),1,8)='AW000001');
```

But the first version, using late-binding of column name, just sounds more natural.

Of course, since it is late-binding, we are not able to let the compiler check at compile time for the column name. If the column name in the source code is wrong, an error will be triggered at runtime only.

First of all, let's see the fastest way of accessing the row content.

In all cases, using the textual version of the column name ('AccountNumber') is slower than using directly the column index. Even if our SynDB.pas library uses a fast lookup using hashing, the following code will always be faster:

```delphi
var
  Customer: Integer;
begin
  with Props.Execute(
'select * from Sales.Customer where AccountNumber like ?', [ AW000001% ], @Customer) do begin
   Customer := ColumnIndex('AccountNumber');
   while Step do
     assert(Copy(ColumnString(Customer), 1, 8) = 'AW000001');
end;
end;

But to be honest, after profiling, most of the time is spend in the Step method, especially in fRowSet.GetData. In practice, I was not able to notice any speed increase worth mentioning, with the code above.

Our name lookup via a hashing function (i.e. TDynArrayHashed) just does its purpose very well.

On the contrary the Ole-Automation based late-binding was found out to be slower, after profiling. In fact, the Row.AccountNumber expression calls an hidden DispInvoke function, which is slow when called multiple times. Our SynCommons.pas unit is able to hack the VCL, and by patching the VCL code in-memory, will call an optimized version of this function. Resulting speed is very close to direct Column['AccountNumber'] call. See SDD # DI-2.2.3.

8.1.9. TDataSet and SynDB

Since our SynDB.pas unit does not rely on the Delphi's DB.pas unit, its result sets do not inherit from the TDataSet.

As a benefit, those result sets will be much faster, when accessed from your object code. But as a drawback, you won't be able to use them in your regular VCL applications.

In order to easily use the SynDB.pas unit with VCL components, you can create TDataSet results sets from any SynDB query.

You have access to two kind of optimized TDataSet result sets:

<table>
<thead>
<tr>
<th>TDataSet class</th>
<th>Operation</th>
<th>Unit</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>TClientDataSet</td>
<td>read write</td>
<td>SynDBMidasVCL.pas</td>
<td>Slow due to memory copy</td>
</tr>
<tr>
<td>TSynSQLStatementDataSet</td>
<td>read only</td>
<td>SynDBVCL.pas</td>
<td>Fast due to direct mapping</td>
</tr>
</tbody>
</table>

You can therefore assign the result of a SynDB request to any TDataSource, as such for our fast TSynSQLStatementDataSet read/only storage:

ds1.DataSet.Free; // release previous TDataSet
ds1.DataSet := ToDataSet(ds1,aProps.Execute('select * from people',[]));

or for a TClientDataSet kind of in-memory storage:

ds1.DataSet.Free; // release previous TDataSet
ds1.DataSet := ToClientDataSet(ds1,aProps.Execute('select * from people',[]));

See sample "17 - TClientDataset use" to find out more about using such TDataSet, including some speed information. You need to have run the TestSQL3.dpr set of regression tests before, to have the expected SQlite3 data file.

8.1.10. TQuery emulation class

The SynDB.pas unit offers a TQuery-like class. This class emulates regular TQuery classes, without inheriting from DB.pas nor its slow TDataSet.

It mimics basic TQuery VCL methods, with the following benefits:
- Does not inherit from TDataSet, but has its own light implementation over SynDB.pas
  ISQLDBStatement result sets, so is usually much faster;
- Will also be faster for field and parameters access by name - or even index;
- Is Unicode-ready, even with older pre-Unicode version of Delphi, able to return the data as
  WideString, independently from the current system charset;
- You can still create a TDataSet from SynDB's TQuery, via the ToDataSet() function defined in
  SynDBVCL.pas.

Of course, since it is not a TDataSet component, you can not use it directly as a regular replacement
for your RAD code.
But if your application is data-centric and tried to encapsulate its business logic with some classes - i.e.
if it tried to properly implement OOP, not RAD - you can still replace directly your existing code with
the TQuery emulator:

```delphi
Q := TQuery.Create(aSQLDBConnection);
try
  Q.SQL.Clear; // optional
  Q.SQL.Add('select * from DOMAIN.TABLE');
  Q.SQL.Add(' WHERE ID_DETAIL=:detail;');
  Q.ParamByName('DETAIL').AsString := '12342002010000430015';
  Q.Open;
  Q.First; // optional
  while not Q.Eof do begin
    assert(Q.FieldByName('id_detail').AsString='12342002010000430015');
    Q.Next;
  end;
  Q.Close; // optional
finally
  Q.Free;
end;
```

You should better use TSQLDBStatement instead of this wrapper, but having such code-compatible
TQuery replacement could make easier some existing code upgrade, especially for Legacy code and
existing projects (page 77).
For instance, it will help to avoid deploying the deprecated BDE, generate (much) smaller executable,
access any database without paying a big fee, avoid rewriting a lot of existing code lines of a big legacy
application, or let your old application communicate with the database over plain HTTP, without the
need to install any RDBMS client - see below (page 263).

8.1.11. Storing connection properties as JSON

You can use a TSynConnectionDefinition storage to persist the connection properties as a JSON
content, in memory or file.

Typical stored content could be:

```json
{
  "Kind": "TSQLDBSQLite3ConnectionProperties",
  "ServerName": "database.db3",
  "DatabaseName": "",
  "UserID": "",
  "Password": "PtvlPA=="
}
```

The "Kind" parameter will be used to store the actual TSQLDBConnectionProperties class name. So
switching from one database to another could be easily done at runtime, by modifying a setting stored
e.g. in a JSON text file, without the need to recompile the application. Note that the SynDB* units
implementing the class should be compiled within the executable, e.g. SynDBSQLite3.pas for
TSQLDBSQLite3ConnectionProperties, or SynDBZeos.pas for TSQLDBZeosConnectionProperties.
To create a new TSQLDBConnectionProperties instance from a local JSON file, you could simply write:

```delphi
var Props: TSQLDBConnectionProperties;
...
Props := TSQLDBConnectionProperties.CreateFromFile('localDBsettings.json');
```

The password will be encrypted and encoded as Base64 in the file, for safety. You could use TSynConnectionDefinition's Password and PasswordPlain properties to compute the value to be written on disk.

Since TSynConnectionDefinition is a TSynPersistent class, you can nest it into a TSynAutoCreateFields instance containing all settings of your application. Then mORMot.pas' ObjectToJSON/ObjectToJSONFile and JSONToObject/JSONFileToObject functions could be used for persistence as a file or in a database, of those global settings.

See also TSQLRest.CreateFrom() for a similar feature at ORM/REST level, and function TSQLRestCreateFrom( aDefinition: TSynConnectionDefinition ) as defined in mORMotDB.pas which is able to create a regular local ORM if aDefinition.Kind is a TSQLRest class name, but also an ORM with external DB storage - see below (page 268) - if aDefinition.Kind is a TSQLDBConnectionProperties class name.
8.2. SynDB clients

From the SynDB.pas logical point of view, here is how databases can be accessed:

Of course, the physical implementation is more complicated, as was stated in SynDB Architecture (page 242).

We will now detail how these available database connections are interfaced as SynDB.pas classes.

8.2.1. OleDB or ODBC to rule them all

OleDB (Object Linking and Embedding, Database, sometimes written as OLE DB or OLE-DB) is an API designed by Microsoft for accessing data from a variety of sources in a uniform manner.

Of course, you have got the Microsoft SQL Native Client to access the MS SQL Server 2005/2008/2012, but Oracle provides a native OleDB provider (even if we found out that this Oracle provider, including the Microsoft's version, have problems with BLOBs). Do not forget about the Advantage Sybase OleDB driver and such...

If you plan to connect to a MS SQL Server, we highly recommend using the T01eDBMSSQL2012ConnectionProperties class, corresponding to SQLNCLI11, part of the Microsoft® SQL Server® 2012 Native Client, it is able to connect to any revision of MS SQL Server (event MS SQL Server 2008), and was found to be the more stable. You can get it from http://www.microsoft.com/en-us/download/details.aspx?id=29065, by downloading the sqlncli.msi corresponding to your Operating System. Most of the time, you should download the X64 Package of sqlncli.msi, which will also install the 32-bit version of SQL Server Native Client, so will work for a 32-bit Delphi executable - the X86 Package is for a 32-bit Windows system only.
**ODBC (Open Database Connectivity)** is a standard C programming language middle-ware API for accessing database management systems (DBMS). ODBC was originally developed by Microsoft during the early 1990s, then was deprecated in favor to OleDB. More recently, Microsoft is officially deprecating OleDB, and urge all developers to switch to the open and cross-platform ODBC API for native connection. Back & worse strategy from Microsoft... one more time! http://blogs.msdn.com/b/sqlnativeclient/archive/2011/08/29/microsoft-is-aligning-with-odbc-for-native-relational-data-access.aspx.

**SynDB and ODBC**

By using our own OleDB and ODBC implementations, we will for instance be able to convert directly the OleDB or ODBC binary rows to JSON, with no temporary conversion into the Delphi high-level types (like temporary string or variant allocations). The resulting performance is much higher than using standard TDataSet or other components, since we will bypass most of the layers introduced by BDE/dbExpress/FireDAC/AnyDAC component sets.

Most OleDB / ODBC providers are free (even maintained by the database owner), others will need a paid license.

It is worth saying that, when used in a mORMot Client-Server architecture, object persistence using an OleDB or ODBC remote access expects only the database instance to be reachable on the Server side. Clients could communicate via standard HTTP, so won't need any specific port forwarding or other IT configuration to work as expected.

### 8.2.2. ZEOS via direct ZDBC

#### 8.2.2.1. The mORMot’s best friend

ZeosLib, aka Zeos, is an Open Source library which provides native access to many database systems, developed for Delphi, Kylix and Lazarus / FreePascal. It is fully object-oriented and with a totally modular design. It connects to the databases by wrapping their native client libraries, and makes them accessible via its abstract layer, named ZDBC. Originally, ZDBC was a port of JDBC 2.0 (Java Database Connectivity API) to Object Pascal. Since that time the API was slightly extended but the main ideas remain unchanged, so official JDBC 2.0 specification is the main entry point to the ZDBC API.

The latest 7.x branch was deeply re-factored, and new methods and performance optimization were introduced. In fact, we worked hand by hand with Michael (the main contributor of ZeosLib) to ensure that the maximum performance is achieved. The result is an impressive synergy of mORMot and ZeosLib, for both reading or writing data.

Since revision 1.18 of the framework, we included direct integration of ZeosLib into mORMot persistence layer, with direct access to the ZDBC layer. That is, our SynDBZeos unit does not reference DB.pas, but access directly to the ZDBC interfaces.
Such direct access, by-passing the VCL DB.pas layer and its TDataSet bottleneck, is very close to our SynDB.pas design. As such, ZeosLib is a first class citizen library for mORMot. The SynDBZeos unit is intended to be a privileged access point to external SQL databases.

### 8.2.2.2. Recommended version

We recommend that you download the 7.2 branch of Zeos/ZDBC, which is the current trunk, at the time of this writing.

A deep code refactoring has been made by the Zeos/ZDBC authors (thanks a lot Michael, aka EgonHugeist!), even taking care of mORMot expectations, to provide the best performance and integration, e.g. for UTF-8 content processing.

In comparison with the previous 7.1 release, speed increase can be of more than 10 times, depending on the database back-end and use case!

When writing data (i.e. Add/Update/Delete operations), Array binding support has been added to the Zeos/ZDBC 7.2 branch, and our SynDBZeos unit will use it if available, detecting if IZDatabaseInfo.SupportsArrayBindings property is true - which will be the case for Oracle and FireBird providers by now. Our ORM benefits from it, when processing in BATCH mode, even letting ZDBC creates the optimized SQL - see below (page 357).

Performance at reading is very high, much higher than any other DB.pas based library, in case of single record retrieval. For instance, TSQLDBZEOSStatement.ColumnsToJSON() will avoid most temporary memory allocation, and is able to create the JSON directly from the low-level ZDBC binary buffers.

If you need to stick to a version prior to 7.2, and want to work as expected with a SQLite3 back-end (but you shouldn't have any reason to do so, since Zeos will be slower compared to SynDBSQLite3), you need to apply some patches for Zeos < 7.2, in methods TZSQLiteCAPIPreparedStatement.ExecuteQueryPrepared() and TZSQLiteResultSet.FreeHandle, as stated as comment at the beginning of SynDBZeos.pas.

### 8.2.2.3. Connection samples

If you want e.g. to connect to MySQL via Zeos/ZDBC, follow those steps:

- Download "Windows (x86, 32-bit), ZIP Archive" from http://dev.mysql.com/downloads/connector/c. - then extract the archive: only libmysql.dll is needed and should be placed either in the executable folder, either in the system PATH;
- Connect as usual e.g. via

```pascal
```
- Or using the URI() method:

```pascal
fConnection := TSQLDBZEOSConnectionProperties.Create(
  TSQLDBZEOSConnectionProperties.URI(dMySQL,'192.168.2.60:3306'), 'root', 'dev');
```

For **PostgreSQL**, the Zeos driver needs only libpq.dll and libintl.dll e.g. from [http://www.enterprisedb.com/products-services-training/pgbindownload.](http://www.enterprisedb.com/products-services-training/pgbindownload.)

```pascal
PropsPostgreSQL := TSQLDBZEOSConnectionProperties.Create(
  TSQLDBZEOSConnectionProperties.URI(dPostgreSQL,'localhost:5432'),
  'dbname', 'username', 'password');
```

You may therefore use the TSQLDBZEOSConnectionProperties.URI() method to compute the expected ZDBC connection string:

```pascal
PropsOracle := TSQLDBZEOSConnectionProperties.Create(
  TSQLDBZEOSConnectionProperties.URI(dOracle,'', 'oci64\oci.dll'),
  'tnsname', 'user', 'pass');
PropsFirebirdEmbedded := TSQLDBZEOSConnectionProperties.Create(
  TSQLDBZEOSConnectionProperties.URI(dFirebird,'', 'Firebird\fbembed.dll'),
  'databasefilename', '', '');
PropsFirebirdRemote := TSQLDBZEOSConnectionProperties.Create(
  TSQLDBZEOSConnectionProperties.URI(dFirebird,'192.168.1.10:3055',
  'c:\Firebird_2_5\bin\fbclient.dll',false),
  '3camadas', 'sysdba', 'masterkey');
```

See TSQLDBZEOSConnectionProperties documentation for further information about the expected syntax, and available abilities of this great open source library.

### 8.2.3. Oracle via OCI

For our framework, and in completion to SynDBZeos or our SynOleDb / SynDBODBC units, the SynDBOracle unit has been implemented. It allows **direct access** to any remote Oracle server, using the **Oracle Call Interface**.

**Oracle Call Interface** (OCI) is the most comprehensive, high performance, native unmanaged interface to the Oracle Database that exposes the full power of the Oracle Database. A direct interface to the oci.dll library was written, using our DB abstraction classes introduced in SynDB.pas.

We tried to implement all best-practice patterns detailed in the official **Building High Performance Drivers for Oracle** reference document.

Resulting speed is quite impressive: for all requests, SynDBOracle is 3 to 5 times faster than a SynOleDb connection using the native **OleDB Provider** supplied by Oracle. A similar (even worse) speed penalty has been observed in comparison with the official ODBC driver from Oracle, via a SynDBODBC-based connection. For more detailed numbers, see **Data access benchmark** (page 199).

### 8.2.3.1. Optimized client library

It is worth saying that, when used in a mORMot Client-Server architecture, object persistence using an Oracle database expects only the Oracle instance to be reachable on the Server side, just like with OleDB or ODBC.

Here are the main features of this SynDBOracle unit:

- **Direct access** to the **Oracle Call Interface** (OCI) client, with no BDE, Midas, DBExpress, nor OleDB / ODBC provider necessary;
- Dedicated to work with **any version** of the Oracle OCI interface, starting from revision 8;
- **Optimized for the latest features** of Oracle 11g/12c (e.g. using native Int64 for retrieving NUMBER fields with no decimal);
- Able to work with the **Oracle Instant Client** for No Setup applications (installation via file/folder copy);
- **Natively Unicode** (uses internal UTF-8 encoding), for all version of Delphi, with special handling of each database char-set;
- Tried to achieve best performance available from every version of the Oracle client;
- Designed to work under any version of Windows, either in 32 or 64-bit architecture (but the OCI library must be installed in the same version than the compiled Delphi application, i.e. only 32-bit for this current version);
- **Late-binding** access to column names, using a new dedicated Variant type (similar to Ole Automation runtime properties);
- Connections are multi-thread ready with low memory and CPU resource overhead;
- Can use connection strings like `'//host[:port]/[service_name]'`, avoiding use of the TNSNAME.ORA file;
- Use Rows Array and BLOB fetching, for best performance (ZEOS/ZDBC did not handle this, for instance);
- Handle Prepared Statements - on both client and server side, if available - server side caching lead to up a 3 times speed boost, from our experiment;
- Implements Array Binding for very fast bulk modifications - insert, update or deletion of a lot of rows at once;
- Implements binding of a TInt64DynArray or TRawUTF8DynArray as parameter, e.g. within a SELECT .. IN where clause;
- **Cursor support**, which is pretty common when working with stored procedures and legacy code.

Of course, this unit is perfectly integrated with the External SQL database access (page 239) process. For instance, it features native export to JSON methods, which will be the main entry point for our ORM framework. And **Array binding** is handled directly during BATCH sequences - see below (page 350).

### 8.2.3.2. Direct connection without Client installation

You can use the latest version of the **Oracle Instant Client** (OIC) provided by Oracle - see [http://www.oracle.com/technetwork/database/features/instant-client..](http://www.oracle.com/technetwork/database/features/instant-client..) - which allows to run client applications without installing the standard (huge) Oracle client or having an ORACLE_HOME.
Oracle Connectivity with SynDBOracle

Just deliver the few dll files in the same directory than the application (probably a mORMot server), and it will work at amazing speed, with all features of Oracle (other stand-alone direct Oracle access library rely on deprecated Oracle 8 protocol).

8.2.3.3. Oracle Wallet support

Password credentials for connecting to databases can now be stored in a client-side Oracle Wallet, a secure software container used to store authentication and signing credentials.

This wallet usage can simplify large-scale deployments that rely on password credentials for connecting to databases. When this feature is configured, application code, batch jobs, and scripts no longer need embedded user names and passwords. Risk is reduced because such passwords are no longer exposed in the clear, and password management policies are more easily enforced without changing application code whenever user names or passwords change.

In order to use this feature, set TSQLDBOracleConnectionProperties.UseWallet to true before connecting to the database.

Wallet configuration is performed on the computer where server is running. You must perform a full Oracle client setup: OIC - see Direct connection without Client Installation (page 257) - does not give access to wallet authentication.

Steps to create a Wallet:

1) Create a folder for you wallet:

   > mkdir c:\OraWallets

2) Create a wallet on the client by using the following syntax at the command line:

   > mkstore -wrl c:\OraWallets -create

   Oracle will ask you for the main wallet password - remember it!

3) Create database connection credentials in the wallet by using the following syntax at the command line:

   mkstore -wrl c:\OraWallets -createCredential TNS_alias_name_from_tnsnames_ora username password

   where password is the password of database user. Oracle will ask you the wallet password - use the main password from previous step.

4) In the client sqlnet.ora file, add the WALLET_LOCATION parameter and set it to the directory location of the wallet and set SQLNET.WALLET_OVERRIDE parameter to TRUE:

   SQLNET.WALLET_OVERRIDE = TRUE
   WALLET_LOCATION =
     (SOURCE =
       (METHOD = FILE)
       (METHOD_DATA =
         (DIRECTORY = c:\OraWallets)
       )
     )

   You can not drop a database while it has a wallet. You need to delete wallet credentials via the next command:

   mkstore -wrl wallet_location -deleteCredential db_alias

   Oracle will ask to enter the wallet password - use the same password which you used during wallet
creation.

Note that there is also an Oracle Wallet Manager tool available with your database distribution, if you prefer to use a GUI tool for database administration. See https://docs.oracle.com/cd/B28359_01/network.111/b28530/asowalet.htm..

### 8.2.4. SQLite3

For our ORM framework, we implemented an efficient SQLite3 wrapper, joining the SQLite3 engine either statically (i.e. within the main exe) or from external sqlite3.dll.

It was an easy task to let the SynSQLite3.pas unit be called from our SynDB.pas database abstract classes. Adding such another Database is just a very thin layer, implemented in the SynDBSQLite3.pas unit.

If you want to link the SQLite3 engine to your project executable, ensure you defined the SynSQLite3Static.pas unit in your uses clause. Otherwise, define a TSQLite3LibraryDynamic instance to load an external sqlite3.dll library:

```pascal
FreeAndNil(sqlite3); // release any previous instance (e.g. static)
sqlite3 := TSQLite3LibraryDynamic.Create;
```

To create a connection property to an existing SQLite3 database file, call the TSQLDBSQLite3ConnectionProperties.Create constructor, with the actual SQLite3 database file as ServerName parameter, and (optionally the proprietary encryption password in Password - available since rev. 1.16); others (DataBaseName, UserID) are just ignored.

These classes will implement an internal statement cache, just as the one used for TSQLRestServerDB. In practice, using the cache can make process up to two times faster (when processing small requests).

When used within the mORMot ORM, you have therefore two ways of accessing the SQLite3 engine:
- Either directly from the ORM core;
- Either virtually, as external tables.

![mORMot ORM, SynDB, mORMot and SQLite3](image)

**SynDB, mORMot and SQLite3**

If your mORMot-based application purpose is to only use one centralized SQLite3 database, it does not make sense to use SynDBSQLite3 external tables. But if you want, in the future, to be able to connect to any external database, or to split your data in several database files, using those external SQLite3 tables do make sense. Of course, the SQLite3 engine library itself will be shared with both internal and external process.

### 8.2.5. DB.pas libraries

Since revision 1.18 of the framework, a new SynDBDataset.pas unit has been introduced, able to
interface any DB.pas based library to our SynDB.pas classes, using TDataset to retrieve the results. Due to the TDataset design, performance is somewhat degraded in respect to direct SynDB.pas connection (e.g. results for SQLite3 or Oracle), but it also opens the potential database access.

Some dedicated providers have been published in the SynDBDataset sub-folder of the mORMot source code repository. Up to now, FireDAC (formerly AnyDAC), UniDAC and BDE libraries are interfaced, and a direct connection to the NexusDB engine is available.

Since there are a lot of potential combinations here - see SynDB Architecture (page 242) - feedback is welcome. Due to our Agile process, we will first stick to the providers we need and use. It is up to mORMot users to ask for additional features, and provide wrappers, if possible, or at least testing abilities. Of course, DBExpress will benefit to be integrated, even if Embarcadero just acquired AnyDAC and revamped/renamed it as FireDAC - to make it the new official platform.

8.2.5.1. NexusDB access

NexusDB is a "royalty-free, SQL:2003 core compliant, Client/Server and Embedded database system, with features that rival other heavily licensed products" (vendor’s terms).

We used and tested the free embedded edition, which is a perfect match for a Client-Server ORM framework like mORMot - see http://www.nexusdb.com/support/index.php?q=FreeEmbedded..

8.2.5.2. FireDAC / AnyDAC library

FireDAC is an unique set of Universal Data Access Components for developing cross platform database applications on Delphi. This was in fact a third-party component set, bought by Embarcadero to DA-SOF Technologies (formerly known as AnyDAC), and included with several editions of Delphi XE3 and up. This is the new official platform for high-speed database development in Delphi, in favor to the now deprecated DBExpress.
Our integration within SynDB.pas units and the mORMot persistence layer has been tuned. For instance, you can have direct access to high-speed FireDAC Array DML feature, via the ORM batch process, via so-called array binding - see below (page 356).

8.2.5.3. UniDAC library

*Universal Data Access Components (UniDAC)* is a cross-platform library of components that provides direct access to multiple databases from Delphi. See [http://www.devart.com/unidac](http://www.devart.com/unidac).

For instance, to access to a MySQL remote database, you should be able to connect using:

```delphi
PropsMySQL := TSQLDBUniDACConnectionProperties.Create(
  TSQLDBUniDACConnectionProperties.URI(dMySQL,'192.168.2.60:3306'),
  'world', 'root', 'dev');
```

This library gives pretty stable results, but lack of the array binding feature, in comparison to *FireDAC*.

8.2.5.4. BDE engine

*Borland Database Engine* (BDE) is the Windows-based core database engine and connectivity software shipped with earlier versions of *Delphi*. Even if it is deprecated, and replaced by DBExpress since 2000, it is a working solution, easy to interface as a SynDB.pas provider.
Please do not use the BDE on any new project!
You should better switch to another access layer.
8.2.6. Remote access via HTTP

The SynDBRemote.pas unit allows you to create database applications that perform SQL operations on a remote HTTP server, instead of a database server. You can create connections just like any other SynDB.pas database, but the transmission will take place over HTTP. As a result, no database client is to be deployed on the end user application: it will just use HTTP requests, even over Internet. You can use all the features of SynDB.pas classes, with the ease of one optimized HTTP connection.

![SynDB Remote access Overview](image)

This feature is not part of our RESTful ORM, so does not use the mORMot.pas unit, but its own optimized protocol, using enhanced security (transmission encryption with user authentication and optional HTTPS) and automatic data compression. Only the HTTP client and server classes, from the SynCrtsSock.pas unit, are used.

Since your application can use both TDataSet - see TDataset and SynDB (page 250) - and emulated TQuery - see TQuery emulation class (page 250), this new mean of transmission may make it easy to convert existing Delphi client-server applications into Multi-tier architecture (page 88) with minimal changes in source code. Then, for your new code, you may switch to a SOA / ORM design, using mORMot's RESTful abilities - see below (page 295).

The transmission protocol uses an optimized binary format, which is compressed, encrypted and digitally signed on both ends, and the remote user authentication will be performed via a challenge validation scheme. You can also publish your server over HTTPS, if needed, in http.sys kernel mode.

8.2.6.1. Server and Client classes

To publish your SynDB.pas connection, you just need to initialize one of the TSQLDBServer* classes defined in SynDBRemote.pas:

```
TSQLDBServerHttpApi
TSQLDBServerSockets
TSQLDBServerAbstract
```
SynDB Remote access Server classes hierarchy

You can define either a HTTP server based on the socket API - TSQLDBServerSockets - or the more stable and fast TSQLDBServerHttpApi class (under Windows only), which uses the http.sys kernel mode HTTP server available since Windows XP - see below (page 326).

For the client side, you could use one of the following classes also defined in SynDBRemote.pas:

- TSQLDBSocketConnectionProperties
- TSQLDBHTTPConnectionPropertiesAbstract
- TSQLDBHttpRequestConnectionProperties
- TSQLDBWinHTTPConnectionProperties
- TSQLDBWinINetConnectionProperties
- TSQLDBCurlConnectionProperties

SynDB Remote access Client classes hierarchy

Note that TSQLDBHttpRequestConnectionProperties is an abstract parent class, so you should not instantiate it directly, but one of its inherited implementations.

As you can see, you may choose between a pure socket API client, others using WinINet or WinHTTP (under Windows), or the libcurl API (especially on Linux). The TSQLDBWinHTTPConnectionProperties class is the more stable over the Internet on Windows, even if plain sockets tend to give better numbers on localhost as stated by our Data access benchmark (page 199). Please read below (page 329) for a comparison of the diverse APIs.

8.2.6.2. Publish a SynDB connection over HTTP

To define a HTTP server, you may write:

```pascal
uses SynDB, // RDBMS core
    SynDBSQLite3, SynSQLite3Static, // static SQLite3 engine
    SynDBRemote; // for HTTP server
...
var Props: TSQLDBConnectionProperties;
    HttpServer: TSQLDBServerAbstract;
...
Props := TSQLDBSQLite3ConnectionProperties.Create('data.db3','','','');
HttpServer := TSQLDBServerHttpApi.Create(Props,'syndbremote','8092','user','pass');
```

The above code will initialize a connection to a local data.db3 SQLite3 database (in the Props variable), and then publish it using the http.sys kernel mode HTTP server to the http://1.2.3.4:8092/syndbremote URI - if the server's IP is 1.2.3.4.

A first user is defined, with 'user' / 'pass' credentials. Note that in our remote access, user management does not match the RDBMS user rights: you should better have your own set of users at application level, for higher security, and a better integration with your business logic. If creating a new user on a RDBMS could be painful, managing remote user authentication is pretty easy on the SynDBRemote.pas side, by using the Protocol.Authenticate property of the server:

```pascal
...
```

You could also share mORMot's REST authentication users below (page 548), by replacing the default TSynAuthentication class instance with TSynAuthenticationRest, as defined in mORMot.pas. Note using at the same time SynDBRemote and mORMot's ORM/Atualization sounds like a weak design, but may...
have its benefits when dealing with legacy code, and a lot of existing SQL statements.

The URI should be registered to work as expected, just as expected by the http.sys API - see below (page 327). You may either run the server once with the system Administrator rights, or call the following method (as we do in TestSQL3Register.dpr) in your setup application:

```pascal
THttpApiServer.AddUrlAuthorize('syndbremote', '8092', false, '+');
```

### 8.2.6.3. SynDB client access via HTTP

On the client side, you can then write:

```pascal
uses SynDB, // RDBMS core
    SynDBRemote; // for HTTP client

var
    Props: TSQLDBConnectionProperties;

... Props := TSQLDBWinHTTPConnectionProperties.Create('1.2.3.4:8092', 'syndbremote', 'user', 'pass');
```

As you can see, there is no link to SynDBSQLite3.pas nor SynSQLite3Static.pas on the client side. Just the HTTP link is needed. No need to deploy the RDBMS client libraries with your application, nor setup the local network firewall.

We defined here a single user, with 'user' / 'pass' credentials, but you may manage more users on the server side, using the `Protocol.Authenticate` property of `TSQLDBServerAbstract`.

Then, you execute your favorite SQL using the connection just as usual:

```pascal
procedure Test(Props: TSQLDBConnectionProperties);
var
    Stmt: ISQLDBRows;
begin
    Stmt := Props.Execute('select * from People where YearOfDeath=?', [1519]);
    while Stmt.Step do begin
        assert(Stmt.ColumnInt('ID') > 0);
        assert(Stmt.ColumnInt('YearOfDeath') = 1519);
    end;
end;
```

Or you may use it with VCL components, using the SynDBVCL.pas unit:

```pascal
d1.DataSet.Free; // release previous TDataSet
d1.DataSet := ToDataSet(d1, Props.Execute('select * from people', []));
```

The `TSynSQLStatementDataSet` result set will map directly the raw binary data returned by the `TSQLDBServer*` class, avoiding any slow data marshalling in your client application, even for huge content. Note that all the whole data is computed and sent by the server: even if you display only the first rows in your `TDBGGrid`, all the data has been transmitted. In fact, partial retrieval works well on a local network, but is not a good idea over the Internet, due to its much higher ping. So consider adding some filter fields, or some application-level paging, to reduce the number of rows retrieved from the `SynDBRemote` server.

If you defined you own `TSynAuthentication` class on the server class (e.g. to use REST users and groups via `TSynAuthenticationRest`), you should create you own class, and override the following method:

```pascal
procedure TSQLDBWinHTTPConnectionPropertiesRest.SetInternalProperties;
begin
    if fProtocol=nil then
        fProtocol := TSQLDBRemoteConnectionProtocol.Create(SynAuthenticationRest.Create(nil, []));
    inherited;
end;
```

This overridden method will inherit from `TSQLDBWinHTTPConnectionProperties` all its behavior, but
use the ORM/SOA authentication scheme for validating its users on the server side.

### 8.2.6.4. Advanced use cases

You may use this remote connection feature e.g. to mutate a stand-alone shared SQLite3 database into a high performance but low maintenance client-server database engine. You may create it as such on the server side:

```pascal
var props: TSQLDBSQLite3ConnectionProperties;
server: TSQLDBServerHttpApi;
...
props := TSQLDBSQLite3ConnectionProperties.Create('database.db3','','','');
props.MainSQLite3DB.Synchronous := smOff;
props.MainSQLite3DB.LockingMode := lmExclusive; // tune the performance
server := TSQLDBServerHttpApi.Create(props,'syndbremote','8092','user','password');
...
```

You could share an existing SQLite3 database instance (e.g. a TSQLRestServerDB used for our RESTful ORM - see Database layer (page 196)) by creating the properties as such:

```pascal
props := TSQLDBSQLite3ConnectionProperties.Create(aRestServerDB.DB);
server := TSQLDBServerHttpApi.Create(props,'syndbremote','8092','user','password');
```

If you use the http.sys kernel-mode server, you could share the same IP port between regular ORM/SOA operations (which may be 80 for a "pure" HTTP server), and remote SynDB access, if the database name (i.e. here 'syndbremote') does not conflict with a ORM table nor a method service.

Note that you can also customize the transmission protocol by setting your own TSQLDBProxyConnectionProtocol class on both server and server sides.

### 8.2.6.5. Integration with SynDBExplorer

Our SynDBExplorer tool is able to publish in one click any SynDB connection as a HTTP server, or connect to it via HTTP. It could be very handy, even for debugging purposes.

To serve an existing database, just connect to it as usual. Then click on the "HTTP Server" button below the table lists (on left side). You can tune the server properties (HTTP port, database name used for URI, user credentials), then click on the "Start" button.

To connect to this remote connection, run another instance of SynDBExplorer. Create a new connection, using "Remote HTTP" as connection type, and set the other options with the values matching the server side, e.g. with the default "localhost:8092" (replacing localhost with the server IP for an access over the network) for the server name, "syndbremote" for the database name, and "synopse" for both user name and password.

You will be able to access the main server instance remotely, just as if the database was accessed via a regular client.

If the server side database is SQLite3, you just mutated this local engine into a true client-server database - you may be amazed by the resulting performance.

### 8.2.6.6. Do not forget the mORMot!

Even if you may be tempted to use such remote access to implement a n-Tier architecture, you should rather use mORMot's Client-Server ORM instead - see below (page 341) - which offers much better client-server integration - due to the Persistence Ignorance pattern of Domain-Driven Design (page 99), a better OOP and SOLID modeling design - see below (page 389), and even higher performance than raw SQL operations - see e.g. below (page 350) or below (page 359). Our little mORMot is not an
ORM on which we added a data transmission layer: it is a full RESTful system, with a true SOA design.

But for integrating some legacy SQL code into a new architecture, SynDBRemote.pas may have its benefits, used in conjunction with mORMot's higher level features.

Note that for cross-platform clients, mORMot's ORM/SOA patterns are a much better approach: do not put SQL in your mobile application, but use services, so that you will not need to re-validate and re-publish the app to the store after any small fix of your business logic!
8.3. SynDB ORM Integration

8.3.1. Code-first or database-first

When working with any Object-Relational Mapping (ORM) (page 92), you have mainly two possibilities:
- Start from scratch, i.e. write your classes and let the ORM create all the database structure, which will reflect directly the object properties - it is also named "code-first";
- Use an existing database, and then define in your model how your classes map the existing database structure - this is the "database-first" option.

Our mORMot framework implements both paths, even if, like for other ORMs, code-first sounds like a more straight option.

8.3.2. Code-first ORM

An external record can be defined as such, as expected by mORMot's ORM:

```pascal
type
  TSQLRecordPeopleExt = class(TSQLRecord)
    private
    fData: TSQLRawBlob;
    fFirstName: RawUTF8;
    fLastName: RawUTF8;
    fYearOfBirth: integer;
    fYearOfDeath: word;
    fLastChange: TModTime;
    fCreatedAt: TCreateTime;
  published
    property FirstName: RawUTF8 index 40 read fFirstName write fFirstName;
    property LastName: RawUTF8 index 40 read fLastName write fLastName;
    property Data: TSQLRawBlob read fData write fData;
    property YearOfBirth: integer read fYearOfBirth write fYearOfBirth;
    property YearOfDeath: word read fYearOfDeath write fYearOfDeath;
    property LastChange: TModTime read fLastChange write fLastChange;
    property CreatedAt: TCreateTime read fCreatedAt write fCreatedAt;
  end;
```

As you can see, there is no difference with an internal ORM class: it inherits from TSQLRecord, but you may want it to inherit from TSQLRecordMany to use ORM implementation via pivot table (page 165) for instance.

The only difference is this index 40 attribute in the definition of FirstName and LastName published properties: this will define the length (in UTF-16 WideChar or UTF-8 bytes) to be used when creating the external field for TEXT column. See above e.g.:

```pascal
property FirstName: RawUTF8
  index 40
  read fFirstName write fFirstName;
```

In fact, SQLite3 does not care about textual field length, but almost all other database engines expect a maximum length to be specified when defining a VARCHAR column in a table. If you do not specify any length in your field definition (i.e. if there is no index ??? attribute), the ORM will create a column with an unlimited length (e.g. varchar(max) for MS SQL Server). In this case, code will work, but performance and disk usage may be highly degraded, since access via a CLOB is known to be notably slower. The only exceptions to this performance penalty are SQLite3 and PostgreSQL, for which the size unlimited TEXT columns are as fast to process than varchar(#).
By default, no check will be performed by the ORM to ensure that the field length is compliant with the column size expectation in the external database. You can use TSQLEndProperties's SetMaxLengthValidatorForTextFields() or SetMaxLengthFilterForTextFields() method to create a validation or filter rule to be performed before sending the data to the external database - see *Filtering and Validating* (page 172).

Here is an extract of the regression test corresponding to external databases:

```pascal
var RExt: TSQLEndPeopleExt;
(...) fProperties := TSQLEndSQLite3ConnectionProperties.Create(SQLITE_MEMORY_DATABASE_NAME,'','','');
VirtualTableExternalRegister(fExternalModel,TSQLEndPeopleExt,fProperties,'PeopleExternal');
aExternalClient := TSQLEndClientDB.Create(fExternalModel,nil,'testExternal.db3',TSQLEndServerDB);
try
  aExternalClient.Server.StaticVirtualTableDirect := StaticVirtualTableDirect;
aExternalClient.Server.CreateMissingTables;
  Check(aExternalClient.Server.CreateSQLMultiIndex(
    TSQLEndPeopleExt,['FirstName','LastName'],false));
  Start := aExternalClient.ServerTimestamp;
  aID := aExternalClient.Add(RExt,true);
  RExt.FillPrepare(aExternalClient,'FirstName=? and LastName=?',
    [RInt.FirstName,RInt.LastName]); // query will use index -> fast :)
  while RExt.FillOne do ...
  (...)
  Updated := aExternalClient.ServerTimestamp;
  aExternalClient.Update(RExt);
  aExternalClient.UnLock(RExt);
  (...)
  aExternalClient.BatchStart(TSQLEndPeopleExt);
  aExternalClient.BatchUpdate(RExt);
  (...)
  aExternalClient.BatchSend(BatchIDUpdate);
  (...)
  aExternalClient.Delete(TSQLEndPeopleExt,i)
  (...) aExternalClient.BatchStart(TSQLEndPeopleExt);
  aExternalClient.BatchDelete(i);
  (...)
  for i := 1 to BatchID[high(BatchID)] do begin
    RExt.fLastChange := 0;
    RExt.CreatedAt := 0;
    RExt.YearOfBirth := 0;
    ok := aExternalClient.Retrieve(i,RExt,false);
    Check(ok=(i and 127<>0),'deletion');
    if ok then begin
      Check(REExt.CreatedAt>=Start);
      Check(REExt.CreatedAt<=Updated);
      if i mod 100=0 then begin
        Check(REExt.YearOfBirth=REExt.YearOfDeath,'Update');
        Check(REExt.LastChange>=Updated);
      end;
    end;
  end;
```

---

Synopsys mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

SAD - mORMot Framework - Rev. 1.18
Page 270 of 2552
As you can see, there is no difference with using the local SQLite3 engine or a remote database engine. From the Client point of view, you just call the usual RESTful CRUD methods, i.e., `Add() Retrieve() Update() Unlock() Delete()` or their faster `Batch*()` revision - and you can even handle advanced methods like a `FillPrepare` with a complex WHERE clause, or `CreateSQLMultiIndex / CreateMissingTables` on the server side.

Even the creation of the table in the remote database (the 'CREATE TABLE...' SQL statement) is performed by the framework when the `CreateMissingTables` method is called, with the appropriate column properties according to the database expectations (e.g., a `TEXT` for SQLite3 will be a NVARCHAR2 field for Oracle).

The resulting table layout on the external database will be the following:

![Table Layout Diagram](image)

The only specific instruction is the global `VirtualTableExternalRegister()` function, which has to be run on the server side (it does not make any sense to run it on the client side, since for the client there is no difference between any tables - in short, the client do not care about storage; the server does).

In order to work as expected, `VirtualTableExternalRegister()` shall be called before `TSQLRestServer.Create` constructor: when the server initializes, the ORM server must know whenever an `internal` or `external` database shall be managed. In the above code, `TSQLRestClientDB.Create()` will instantiate its own embedded `TSQLRestServerDB` instance.

Note that the `TSQLRecordExternal.LastChange` field was defined as a `TModTime`: in fact, the current date and time will be stored each time the record is updated, i.e., for each `aExternalClient.Add` or `aExternalClient.Update` calls. This is tested by both `RExt.LastChange>=Start` and `RExt.LastChange<=Updated` checks in the latest loop. The time used is the "server-time", i.e., the current time and date on the server (not on the client), and, in the case of external databases, the time of the remote server (it will execute e.g., a `select getdate()` under MS SQL to synchronize the date to be inserted for `LastChange`). In order to retrieve this server-side time stamp, we use `Start := aExternalClient.ServerTimestamp` instead of the local `TimeLogNow` function.

A similar feature is tested for the `CreatedAt` published field, which was defined as `TCreateTime`: it will be set automatically to the current server time at record creation (and not changed on modifications). This is the purpose of the `RExt.CreatedAt<=Updated` check in the above code.

### 8.3.3. Database-first ORM

As we have just seen, the following line initializes the ORM to let `TSQLRecordPeopleExt` data be
accessed via SQL, over an external database connection `fProperties`:

```delphi
VirtualTableExternalRegister(fExternalModel,TSQLRecordPeopleExt,fProperties,'PeopleExternal');
```

We also customized the name of the external table, from its default 'PeopleExt' (computed by trimming TSQLRecord prefix from TSQLRecordPeopleExt) into 'PeopleExternal'.

In addition to table name mapping, the ORM is also able to map the TSQLRecord published properties names to any custom database column name. It is in fact very common that most tables on existing databases to not have very explicit column naming, which may sounds pretty weird when mapped directly as TSQLRecord property names. Even the primary keys of your existing database won't match the ORM's requirement of naming it as ID. All this should be setup as expected.

By default, for a code-driven approach, internal property names will match the external table column names - see TSQLRecordPeopleExt Code-First Field/Column Mapping (page 270)

You can customize this default mapping, writing e.g.

```delphi
fProperties := TSQLDBSQLite3ConnectionProperties.Create(SQLITE_MEMORY_DATABASE_NAME,'','','');
VirtualTableExternalRegister(fExternalModel,TSQLRecordPeopleExt,fProperties,'PeopleExternal');

fExternalModel.Props[TSQLRecordPeopleExt].ExternalDB.
  MapField('ID','Key');
  MapField('YearOfDeath','YOD');

(...) // the remaining code stays the same
```

As an alternative, you can use the VirtualTableExternalMap function and its fluent interface:

```delphi
fProperties := TSQLDBSQLite3ConnectionProperties.Create(SQLITE_MEMORY_DATABASE_NAME,'','','');
VirtualTableExternalMap(fExternalModel,TSQLRecordPeopleExt,fProperties,'PeopleExternal');

fExternalModel.Props[TSQLRecordPeopleExt].ExternalDB.
  MapField('ID','Key');
  MapField('YearOfDeath','YOD');
```

Then you use your TSQLRecordPeopleExt table as usual from Delphi code, with ID and YearOfDeath fields.

But, under the hood, the mORMot ORM will do the mapping when creating all needed SQL statements:
- The "internal" TSQLRecord class will be stored within the PeopleExternal external table;
- The "internal" TSQLRecord.ID field will be an external "Key: INTEGER" column;
- The "internal" TSQLRecord.YearOfDeath field will be an external "YOD: INTEGER" column;
- Other internal published properties will be mapped by default with the same name to external column.

The resulting mapping will therefore be the following:

```
<table>
<thead>
<tr>
<th>TSQLRecordPeopleExt Database-First Field/Column Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID : TID</td>
</tr>
<tr>
<td>Data : TSQLRawBlob</td>
</tr>
<tr>
<td>FirstName : RawUTF8</td>
</tr>
<tr>
<td>LastName : RawUTF8</td>
</tr>
<tr>
<td>YearOfBirth : integer</td>
</tr>
<tr>
<td>YearOfDeath : word</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Key : INTEGER</td>
</tr>
<tr>
<td>Data : BLOB</td>
</tr>
<tr>
<td>FirstName : NVARCHAR(40)</td>
</tr>
<tr>
<td>LastName : NVARCHAR(40)</td>
</tr>
<tr>
<td>YearOfBirth : INTEGER</td>
</tr>
<tr>
<td>YOD : INTEGER</td>
</tr>
</tbody>
</table>
```

Note that only the ID and YearOfDeath column names were customized.

Due to the design of SQLite3 virtual tables, and mORMot internals in its current state, the database primary key must be an INTEGER field to be mapped as expected by the ORM. But you can specify any secondary key, e.g. a TEXT field, via stored AS_UNIQUE definition in code.
8.3.4. Sharing the database with legacy code

It is pretty much possible that you will have to maintain and evolve a legacy project, based on an existing database, with a lot of already written SQL statements - see Legacy code and existing projects (page 77). For instance, you would like to use mORMot for new features, and/or add mobile or HTML clients - see below (page 481).

In this case, the ORM advanced features - like ORM Cache (page 174) or BATCH process, see below (page 350) - may conflict with the legacy code, for the tables which may have to be shared. Here are some guidelines when working on such a project.

To be exhaustive about this question, we need to consider each ORM CRUD operation. We may have to divide them in three kinds: read queries, insertions, and modifications of existing data.

About ORM read queries, i.e. Retrieve() methods, the ORM cache can be tuned per table, and you will definitively lack of some cache, but remember:
- That you can set a "time out" period for this cache, so that you may still benefit of it in most cases;
- That you have a cache at server level and another at client level, so you can tune it to be less aggressive on the client, for instance;
- That you can tune the ORM cache per ID, so some items which are not likely to change can still be cached.

About ORM insertions, i.e. Add() or BatchAdd() methods, when using the external engine, if any external process is likely to INSERT new rows, ensure you set the TSQLRestStorageExternal EngineAddUseSelectMaxID property to TRUE, so that it will compute the next maximum ID by hand. But it still may be an issue, since the external process may do an INSERT during the ORM insertion.

So the best is perhaps to NOT use the ORM Add() or BatchAdd() methods, but rely on dedicated INSERT SQL statement, e.g. hosted in an interface-based service on the server side.

About ORM modifications, i.e. Update() Delete() BatchUpdate() BatchDelete() methods, they sound safe to be used in conjunction with external process modifying the DB, as soon as you use transactions to let the modifications be atomic, and won't conflict any concurrent modifications in the legacy code.

Perhaps the safer pattern, when working with external tables which are to be modified in the background by some legacy code, may be to by-pass those ORM methods, and define server-side interface-based services - see below (page 419). Those services may contain manual SQL, instead of using the ORM "magic". But it will depend on your business logic, and you will fail to benefit from the ORM features of the framework.

Nevertheless, introducing Service-Oriented Architecture (SOA) (page 90) into your application will be very beneficial: ORM is not mandatory, especially if you are "fluent" in SQL queries, know how to make them as standard as possible, and have a lot of legacy code, perhaps with already tuned SQL statements.

Introducing SOA is mandatory to interface new kind of clients to your applications, like mobile apps or AJAX modern sites. To be fair, you should not access directly the database any more, as you did with your legacy Delphi and RAD DB components.

All new features, involving new tables to store new data, will still benefit of the mORMot's ORM, and could still be hosted in the very same external database, shared by your existing code. Then, you will be able to identify seams - see Legacy code and existing projects (page 77) - in your legacy code, and move them to your new mORMot services, then let your application evolve into a newer SOA/MVC architecture, without breaking anything, nor starting from scratch.
8.3.5. Auto-mapping of SQL conflictual field names

If your application is likely to be run on several databases, it may be difficult to handle any potential field name conflict, when you switch from one engine to another. The ORM allows you therefore to ensure that no field name will conflict with a SQL keyword of the underlying database.

In code-first mode, you can use the following method to ensure that no such conflict occurs:

```pascal
fExternalModel.Props[TSQLRecordPeopleExt].ExternalDB.MapAutoKeywordFields;
```

For a database-first database, the following syntax is to be used so that field names will be checked:

```pascal
fExternalModel.Props[TSQLRecordPeopleExt].ExternalDB. // custom field mapping
MapField('ID', 'Key').
MapField('YearOfDeath', 'YOD').
MapAutoKeywordFields;
```

or, if you want to full fluent interface definition:

```pascal
VirtualTableExternalMap(fExternalModel,TSQLRecordPeopleExt,fProperties,'PeopleExternal').
MapField('ID', 'Key').
MapField('YearOfDeath', 'YOD').
MapAutoKeywordFields
```

It is a good idea to call the MapAutoKeywordFields method after any manual field mapping for a database-first database, since even your custom field names may conflict with a SQL keyword.

If any field name is likely to conflict with a SQL keyword, it will be mapped with a trailing '_'. For instance, a 'Select' published property will be mapped into a SELECT_ column in the table.

Even if this option is disabled by default, a warning message will appear in the log proposing to use this MapAutoKeywordFields method, and will help you to identify such issues.

8.3.6. External database ORM internals

The mORMotDB.pas unit implements Virtual Tables access for any SynDB.pas-based external database for the framework.

In fact, this feature will use TSQLRestStorageExternal, TSQLVirtualTableCursorExternal and TSQLVirtualTableExternal classes, defined as such:

```
TSQLRecordVirtual
TSQLRecord
TSQLRecordVirtualTableAutoID
TSQLVirtualTableCursorExternal
TSQLVirtualTableCursor
TSQLVirtualTableExternal
TSQLVirtualTable
```

**External Databases classes hierarchy**

The registration of the class is done by a call to the following new global procedure:

```pascal
procedure VirtualTableExternalRegister(aModel: TSQLModel; aClass: TSQLRecordClass;
aExternalDB: TSQLDBConnectionProperties; const aExternalTableName: RawUTF8);
```

This procedure will register on the Server-side an external database for an ORM class:
- It will define the supplied class to behave like a TSQLRecordVirtualTableAutoID class (i.e. its TSQLModelRecordProperties.Kind property will be overwritten to rCustomAutoID in this ORM
model;
- It will associate the supplied class with a TSQLVirtualTableExternal module;
- The TSQLDBConnectionProperties instance should be shared by all classes, and released globally when the ORM is no longer needed;
- The full table name, as expected by the external database, should be provided here (SQLTableName will be used internally as table name when called via the associated SQLite3 Virtual Table) - if no table name is specified (""), will use SQLTableName (e.g. 'Customer' for a class named TSQLCustomer);
- Internal adjustments will be made to convert SQL on the fly from internal ORM representation into the expected external SQL format (e.g. table name or ID property) - see TSQLRestStorage. AdaptSQLForEngineList method.

Typical usage may be for instance:

```pascal
aProps := TOleDBMSSQLConnectionProperties.Create('.\SQLEXPRESS', 'AdventureWorks2008R2', '', '');
aModel := TSQLModel.Create([TSQLCustomer], 'root');
VirtualTableExternalRegister(aModel, TSQLCustomer, aProps, 'Sales.Customer');
aServer := TSQLRestServerDB.Create(aModel, 'application.db', true);
```

All the rest of the code will use the "regular" ORM classes, methods and functions, as stated by Object-Relational Mapping (page 130).

In order to be stored in an external database, the ORM records can inherit from any TSQLRecord class. Even if this class does not inherit from TSQLRecordVirtualTableAutoID, it will behave as such, once VirtualTableExternalRegister function has been called for the given class.

As with any regular TSQLRecord classes, the ORM core will expect external tables to map an Integer ID published property, auto-incremented at every record insertion. Since not all databases handle such fields - e.g. Oracle - auto-increment will be handled via a select max(id) from tablename statement run at initialization, then computed on the fly via a thread-safe cache of the latest inserted RowID.

You do not have to know where and how the data persistence is stored. The framework will do all the low-level DB work for you. And thanks to the Virtual Table feature of SQLite3, internal and external tables can be mixed within SQL statements. Depending on the implementation needs, classes could be persistent either via the internal SQLite3 engine, or via external databases, just via a call to VirtualTableExternalRegister() before server initialization.

In fact, TSQLVirtualTableCursorExternal1 will convert any query on the external table into a proper optimized SQL query, according to the indexes existing on the external database. TSQLVirtualTableExternal1 will also convert individual SQL modification statements (like insert / update / delete) at the SQLite3 level into remote SQL statements to the external database.

Most of the time, all RESTful methods (GET/POST/PUT/DELETE) will be handled directly by the TSQLRestStorageExternal1 class, and won't use the virtual table mechanism. In practice, most access to the external database will be as fast as direct access, but the virtual table will always be ready to interpret any cross-database complex request or statement.

Direct REST access will be processed as following - when adding an object, for instance:
ORM Access Via REST
Indirect access via virtual tables will be processed as following:

```
TSQLRestServerDB.Add
   (internal or external table)

TSQLRestServerDB.EngineAdd

TSQLRequest
   (internal engine)

SQLite3 engine
   (internal or external table)

SQLite3 file

TSQLVirtualTableExternal.Insert

   (insert INTO...)

ISQLDBStatement

   (ODBC/ZDBC/OleDB...)

External DB client

External DB server
```

**ORM Access Via Virtual Table**

About speed, here is an extract of the test regression log file (see code above, in previous paragraph), which shows the difference between RESTful call and virtual table call, working with more than 11,000 rows of data:

- External via REST: 133,666 assertions passed  409.82ms
- External via virtual table: 133,666 assertions passed  1.12s

The first run is made with TSQLRestServer.StaticVirtualTableDirect set to TRUE (which is the default) - i.e. it will call directly TSQLRestStorageExternal for RESTful commands, and the second will set this property to FALSE - i.e. it will call the SQLite3 engine and let its virtual table mechanism convert it into another SQL calls.

It is worth saying that this test is using an in-memory SQLite3 database (i.e. instantiated via SQLITE_MEMORY_DATABASE_NAME as pseudo-file name) as its external DB, so what we test here is mostly the ORM overhead, not the external database speed. With real file-based or remote databases
(like MS SQL), the overhead of remote connection won't make noticeable the use of Virtual Tables.

In all cases, letting the default `StaticVirtualTableDirect=true` will ensure the best possible performance. As stated by *Data access benchmark* (page 199), using a virtual or direct call won't affect the CRUD operation speed: it will by-pass the virtual engine whenever possible.

### 8.3.7. Tuning the process

Multi-threading abilities of the server, and all available settings, will be detailed below (page 335).

By default, all ORM read operations will be run in concurrent mode, and all ORM write operations will be executed in blocking mode. This is expected to be both safe and fast, with our internal SQLite3 engine, or most of the external databases. But you may need to change this default behavior, depending on the external engine you are connected to.

Most TSQLDBConnectionProperties will inherit from TSQLDBConnectionPropertiesThreadSafe, so will create one connection per thread. This is efficient, but some providers may have issues with it.

First of all, some database client libraries may not allow transactions to be shared among several threads - for instance MS SQL. Other clients may consume a lot of resources for each connection, or may not have good multi-thread scaling abilities. Some database servers do fork their process for each connected client - for instance PostgreSQL: you may want to reduce the server resources by using only one connection, so only one process on the server. To avoid such problems, you can force all ORM write operations to be executed in a dedicated thread, i.e. by setting `amMainThread` (which is not very opportune on a server without UI), or even better via `amBackgroundThread` or a `amBackgroundORMSharedThread`:

```
  aServer.AcquireExecutionMode[execORMWrite] := amBackgroundThread;
```

Secondly, especially on a long-running n-Tier mORMot server, you may suffer from broken connection exceptions. For instance, after a night without any activity, the attempts to access to the external database may fail in the morning, since the connection may have been disconnected by the database server in the meanwhile.

You can use the TSQLDBConnectionProperties.ConnectionTimeOutMinutes property to specify a maximum period of inactivity after which all connections will be flushed and recreated, to avoid potential broken connections issues.

In practice, recreating the connections after a while is safe and won't slow done the process - on the contrary, it may help reducing the consumed resources, and stabilize long running n-Tier servers.

ThreadSafeConnection method will check for the last activity on its TSQLDBConnectionProperties instance, and then call ClearConnectionPool to release all active connections if the idle time elapsed was too long.

As a consequence, if you use this ConnectionTimeOutMinutes property, you should ensure that no other connection is still active on the background, otherwise some unexpected issues may occur.

For instance, you should ensure that your mORMot ORM server runs all its statements in blocking mode for both read and write:

```
  aServer.AcquireExecutionMode[execORMGet] := am***;
  aServer.AcquireExecutionMode[execORMWrite] := am***;
```

Here above, safe blocking `am***` modes are any mode but `amUnlocked`, i.e. either `amLocked`, `amBackgroundThread`, `amBackgroundORMSharedThread` or `amMainThread`. 
9. External NoSQL database access

Our ORM RESTful framework is able to access not only regular SQL database engines, via *SynDB direct RDBMS access* (page 241), but also NoSQL engines - see *NoSQL and Object-Document Mapping (ODM)* (page 96).

Remember the diagram introducing *mORMot’s Database layer* (page 196):

![mORMot Persistence Layer Architecture](image-url)
The following NoSQL engines can be accessed from mORMot's Object Document Mapping (ODM) abilities:

<table>
<thead>
<tr>
<th>NoSQL Engine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TObjectList</td>
<td>In memory storage, with JSON or binary disk persistence</td>
</tr>
<tr>
<td><strong>MongoDB</strong></td>
<td>#1 NoSQL database engine</td>
</tr>
</tbody>
</table>

We can in fact consider our TSQLRestStorageInMemory instance, and its TObjectList storage, as a NoSQL very fast in-memory engine, written in pure Delphi. See In-Memory "static" process (page 230) for details about this feature.

MongoDB (from "humongous") is a cross-platform document-oriented database system, and certainly the best known NoSQL database. According to http://db-engines.com.. in December 2015, MongoDB is at 4th place of the most popular types of database management systems, and at first place for NoSQL database management systems. Our mORMot gives premium access to this database, featuring full NoSQL and Object-Document Mapping (ODM) (page 96) abilities to the framework.

Integration is made at two levels:
- Direct low-level access to the MongoDB server, in the SynMongoDB.pas unit;
- Close integration with our ORM (which becomes defacto an ODM), in the mORMotMongoDB.pas unit.

MongoDB eschews the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas (MongoDB calls the format BSON), which match perfectly mORMot's RESTful approach.

### 9.1. SynMongoDB client

The SynMongoDB.pas unit features direct optimized access to a MongoDB server.

It gives access to any BSON data, including documents, arrays, and MongoDB's custom types (like ObjectId, dates, binary, regex, Decimal128 or Javascript):
- For instance, a TBSONObjectID can be used to create some genuine document identifiers on the client side (MongoDB does not generate the IDs for you: a common way is to generate unique IDs on the client side);
- Generation of BSON content from any Delphi types (via TBSONWriter);
- Fast in-place parsing of the BSON stream, without any memory allocation (via TBSONElement);
- A TBSONVariant custom variant type, to store MongoDB's custom type values;
- Interaction with the SynCommons.pas' TDocVariant custom variant type (page 112) as document storage and late-binding access;
- Marshalling BSON to and from JSON, with the MongoDB extended syntax for handling its custom types.

This unit defines some objects able to connect and manage databases and collections of documents on any MongoDB servers farm:
- Connection to one or several servers, including secondary hosts, via the TMongoClient class;
- Access to any database instance, via the TMongoDatabase class;
- Access to any collection, via the TMongoCollection class;
- It features some nice abilities about speed, like BULK insert or delete mode, and explicit Write Concern settings.
At collection level, you can have direct access to the data, with high level structures like TDocVariant/TBSONVariant, with easy-to-read JSON, or low level BSON content. You can also tune most aspects of the client process, e.g. about error handling or write concerns (i.e. how remote data modifications are acknowledged).

### 9.1.1. Connecting to a server

Here is some sample code, which is able to connect to a MongoDB server, and returns the server time:

```pascal
var Client: TMongoClient;
DB: TMongoDatabase;
serverTime: TDateTime;
res: variant; // we will return the command result as TDocVariant
errmsg: RawUTF8;
begin
  Client := TMongoClient.Create('localhost',27017);
  try
    DB := Client.Database['mydb'];
    writeln('Connecting to ',DB.Name); // will write 'mydb'
    errmsg := DB.RunCommand('hostInfo',res); // run a command
    if errmsg<>'' then exit; // exit on any error
    serverTime := res.system.currentTime;
    writeln('Server time is ',DateTimeToStr(serverTime));
  finally
    Client.Free; // will release the DB instance
  end;
end;
```

Note that for this low-level command, we used a TDocVariant, and its late-binding abilities.

In fact, if you put your mouse over the res variable during debugging, you will see the following JSON content:

```
{
  "system":{
    "currentTime":"2014-05-06T15:24:25",
    "hostname":"Acer",
    "cpuAddrSize":64,
    "memSizeMB":3934,
    "numCores":4,
    "cpuArch":"x86_64",
    "numaEnabled":false,
    "os":{
      "type":"Windows",
      "name":"Microsoft Windows 7",
      "version":"6.1 SP1 (build 7601)"
    },
    "extra":{
      "pageSize":4096,
      "ok":1
    }
  }
}
```

And we simply access to the server time by writing res.system.currentTime.

Here connection was made anonymously. It will work only if the mongod instance is running on the same computer. Safe remote connection, including user authentication, could be made via the TMongoClient.OpenAuth() method: it supports the latest SCRAM-SHA-1 challenge-response mechanism (supported since MongoDB 3.x), or the deprecated MONGODB-CR (for older versions).

```pascal
... Client := TMongoClient.Create('localhost',27017);
try
  DB := Client.OpenAuth('mydb','mongouser','mongopwd');
...```

For safety reasons, never let a MongoDB server be remotely accessible without proper authentication, as stated by [http://docs.mongodb.org/manual/administration/security-access-control](http://docs.mongodb.org/manual/administration/security-access-control). The TMongoDatabase.CreateUser(), CreateUserForThisDatabase() and DropUser() methods allow to easily manage credentials from your applications.

### 9.1.2. Adding some documents to the collection

We will now explain how to add documents to a given collection.

We assume that we have a DB: TMongoDatabase instance available. Then we will create the documents with a TDocVariant instance, which will be filled via late-binding, and via a doc.Clear
pseudo-method used to flush any previous property value:

```pascal
var Coll: TMongoCollection;
  doc: variant;
  i: integer;
begin
  Coll := DB.CollectionOrCreate[COLL_NAME];
  TDocVariant.New(doc);
  for i := 1 to 10 do
    begin
      doc.Clear;
      doc.Name := 'Name ' + IntToStr(i+1);
      doc.Number := i;
      Coll.Save(doc);
      writeln('Inserted with _id=' , doc._id);
    end;
end;
```

Thanks to TDocVariant late-binding abilities, code is pretty easy to understand and maintain.

This code will display the following on the console:

```
Inserted with _id=5369029E4F901EE8114799D9
Inserted with _id=5369029E4F901EE8114799DA
Inserted with _id=5369029E4F901EE8114799DB
Inserted with _id=5369029E4F901EE8114799DC
Inserted with _id=5369029E4F901EE8114799DD
Inserted with _id=5369029E4F901EE8114799DE
Inserted with _id=5369029E4F901EE8114799DF
Inserted with _id=5369029E4F901EE8114799E0
Inserted with _id=5369029E4F901EE8114799E1
Inserted with _id=5369029E4F901EE8114799E2
```

It means that the Coll.Save() method was clever enough to understand that the supplied document does not have any _id field, so will compute one on the client side before sending the document data to the MongoDB server.

We may have written:

```pascal
for i := 1 to 10 do
  begin
    doc.Clear;
    doc._id := ObjectID;
    doc.Name := 'Name ' + IntToStr(i+1);
    doc.Number := i;
    Coll.Save(doc);
    writeln('Inserted with _id=', doc._id);
  end;
end;
```

Which will compute the document identifier explicitly before calling Coll.Save().

In this case, we may have called directly Coll.Insert(), which is somewhat faster.

Note that you are not obliged to use a MongoDB ObjectID as identifier. You can use any value, if you are sure that it will be genuine. For instance, you can use an integer:

```pascal
for i := 1 to 10 do
  begin
    doc.Clear;
    doc._id := i;
    doc.Name := 'Name ' + IntToStr(i+1);
    doc.Number := i;
    Coll.Insert(doc);
    writeln('Inserted with _id=', doc._id);
  end;
end;
```
The console will display now:

```
Inserted with _id=1
Inserted with _id=2
Inserted with _id=3
Inserted with _id=4
Inserted with _id=5
Inserted with _id=6
Inserted with _id=7
Inserted with _id=8
Inserted with _id=9
Inserted with _id=10
```

Note that the mORMot ORM will compute a genuine series of integers in a similar way, which will be used as expected by the TSQLRecord.ID primary key property.

The TMongoCollection class can also write a list of documents, and send them at once to the MongoDB server: this BULK insert mode - close to the Array Binding feature of some SQL providers, and implemented in our SynDB.pas classes - see below (page 350) - can increase the insertion by a factor of 10 times, even when connected to a local instance: imagine how much time it may save over a physical network!

For instance, you may write:

```pascal
var docs: TVariantDynArray;
...
SetLength(docs,COLL_COUNT);
for i := 0 to COLL_COUNT-1 do begin
  TDocVariant.New(docs[i]);
  docs[i].id := ObjectId; // compute new ObjectId on the client side
  docs[i].Name := 'Name ' + IntToStr(i+1);
  docs[i].FirstName := 'FirstName ' + IntToStr(i+COLL_COUNT);
  docs[i].Number := i;
end;
Coll.Insert(docs); // insert all values at once
```

You will find out later for some numbers about the speed increase due to such BULK insert.

### 9.1.3. Retrieving the documents

You can retrieve the document as a TDocVariant instance:

```pascal
var doc: variant;
...
  doc := Coll.FindOne(5);
  writeln('Name: ',doc.Name);
  writeln('Number: ',doc.Number);
```

Which will write on the console:

```
Name: Name 6
Number: 5
```

You have access to the whole Query parameter, if needed:

```pascal
doc := Coll.FindDoc('_id:5',[],
  doc := Coll.FindOne(5); // same as previous
```

This Query filter is similar to a WHERE clause in SQL. You can write complex search patterns, if needed - see [http://docs.mongodb.org/manual/reference/method/db.collection.find..](http://docs.mongodb.org/manual/reference/method/db.collection.find..) for reference.

You can retrieve a list of documents, as a dynamic array of TDocVariant:

```pascal
var docs: TVariantDynArray;
...
Coll.FindDocs(docs);
for i := 0 to high(docs) do
  writeln('Name: ',docs[i].Name, ' Number: ',docs[i].Number);

Which will output:

Name: Name 2 Number: 1
Name: Name 3 Number: 2
Name: Name 4 Number: 3
Name: Name 5 Number: 4
Name: Name 6 Number: 5
Name: Name 7 Number: 6
Name: Name 8 Number: 7
Name: Name 9 Number: 8
Name: Name 10 Number: 9
Name: Name 11 Number: 10

In a GUI application, you could fill a VCL grid using a TDocVariantArrayDataSet as defined in SynVirtualDataSet.pas, for instance:

ds1.DataSet.Free; // release previous TDataSet
ds1.DataSet := ToDataSet(self,FindDocs('{name:?,age:{$gt:?}}','[John'],21],null));

This overloaded FindDocs() method takes a query filter as JSON and parameters (following the MongoDB syntax), and a Projection mapping (null to retrieve all properties). Its returns a TVariantDynArray result, which was mapped to an optimized read-only TDataSet using the overloaded ToDataSet() function. So in our case, the DB grid has been filled with all people named 'John', with age greater than 21.

If you want to retrieve the documents directly as JSON, we can write:

var json: RawUTF8;
...
json := Coll.FindJSON(null,null);
writeln(json);
...

This will append the following to the console:

[[{
    '_id':1,"Name":"Name 2","Number":1},
    {
    '_id':2,"Name":"Name 3","Number":2},
    {
    '_id':3,"Name":"Name 4","Number":3},
    {
    '_id':4,"Name":"Name 5","Number":4},
    {
    '_id':5,"Name":"Name 6","Number":5},
    {
    '_id':6,"Name":"Name 7","Number":6},
    {
    '_id':7,"Name":"Name 8","Number":7},
    {
    '_id':8,"Name":"Name 9","Number":8},
    {
    '_id':9,"Name":"Name 10","Number":9}]]

You can note that FindJSON() has two properties, which are the Query filter, and a Projection mapping (similar to the column names in a SELECT col1,col2).

So we may have written:

json := Coll.FindJSON('{_id:?}',[5]);
writeln(json);

Which outputs:

[{
    '_id':5,"Name":"Name 6","Number":5}]

We used here an overloaded FindJSON() method, which accept the MongoDB extended syntax (here, the field name is unquoted), and parameters as variables.

We can specify a projection:

json := Coll.FindJSON('{_id:?}',[5],'{Name:1}');
writeln(json);

Which will only return the "Name" and "_id" fields (since _id is, by MongoDB convention, always returned:

[{
    '_id':5,"Name":"Name 6"}]}
To return only the "Name" field, you can specify '_id:0,Name:1' as JSON in extended syntax for the *projection* parameter.

```json
["Name":"Name 6"]
```

There are other methods able to retrieve data, also directly as BSON binary data. They will be used for best speed e.g. in conjunction with our ORM, but for most end-user code, using TDocVariant is safer and easier to maintain.

### 9.1.3.1. Updating or deleting documents

The TMongoCollection class has some methods dedicated to alter existing documents.

At first, the Save() method can be used to update a document which has been first retrieved:

```pascal
doc := Coll.FindOne(5);
doc.Name := 'New!';
Coll.Save(doc);
writeln('Name: ',Coll.FindOne(5).Name);
```

Which will write:

```
Name: New!
```

Note that we used here an integer value (5) as key, but we may use an *ObjectId* instead, if needed.

The Coll.Save() method could be changed into Coll.Update(), which expects an explicit Query operator, in addition to the updated document content:

```pascal
doc := Coll.FindOne(5);
doc.Name := 'New!';
Coll.Update(BSONVariant(["_id":5]),doc);
writeln('Name: ',Coll.FindOne(5).Name);
```

Note that by *MongoDB*’s design, any call to Update() will *replace* the whole document.

For instance, if you write:

```pascal
writeln('Before: ',Coll.FindOne(3));
Coll.Update('{}','Name: New Name!');
writeln('After:  ',Coll.FindOne(3));
```

Then the Number field will disappear!

Before: `{"_id":3,"Name":"Name 4","Number":3}`

After: `{"_id":3,"Name":"New Name!"}`

If you need to update only some fields, you will have to use the $set modifier:

```pascal
writeln('Before: ',Coll.FindOne(4));
Coll.Update('{}','{$set:{Name:"New Name!"}}');
writeln('After:  ',Coll.FindOne(4));
```

Which will write on the console the value as expected:

Before: `{"_id":4,"Name":"Name 5","Number":4}`

After: `{"_id":4,"Name":"New Name!","Number":4}`

Now the Number field remains untouched.

You can also use the Coll.UpdateOne() method, which will update the supplied fields, and leave the non specified fields untouched:

```pascal
writeln('Before: ',Coll.FindOne(2));
Coll.UpdateOne(2,_Obj(['Name','NEW']));
writeln('After:  ',Coll.FindOne(2));
```

Which will output as expected:
You can refer to the documentation of the SynMongoDB.pas unit, to find out all functions, classes and methods available to work with MongoDB.

Some very powerful features are available, including Aggregation (available since MongoDB 2.2), which offers a good alternative to standard Map/Reduce pattern. See http://docs.mongodb.org/manual/reference/command/aggregate.. for reference.

9.1.4. Write Concern and Performance

You can take a look at the MongoDBTests.dpr sample - located in the SQLite3\Samples\24 - MongoDB sub-folder of the source code repository, and the TTestDirect classes, to find out some performance information.

In fact, this TTestDirect is inherited twice, to run the same tests with diverse write concern:

```
procedure TTestDirect.ConnectToLocalServer;
...
  fClient := TMongoClient.Create('localhost', 27017);
  if ClassType = TTestDirectWithAcknowledge then
    fClient.WriteConcern := wcAcknowledged else
  if ClassType = TTestDirectWithoutAcknowledge then
    fClient.WriteConcern := wcUnacknowledged;
...
```

wcAcknowledged is the default safe mode: the MongoDB server confirms the receipt of the write operation. Acknowledged write concern allows clients to catch network, duplicate key, and other errors. But it adds an additional round-trip from the client to the server, and wait for the command to be finished before returning the error status: so it will slow down the write process.

With wcUnacknowledged, MongoDB does not acknowledge the receipt of write operation. Unacknowledged is similar to errors ignored; however, drivers attempt to receive and handle network errors when possible. The driver's ability to detect network errors depends on the system's networking configuration.

The speed difference between the two is worth mentioning, as stated by the regression tests status, running on a local MongoDB instance:

1. Direct access

1.1. Direct with acknowledge:
- Connect to local server: 6 assertions passed 4.72ms
- Drop and prepare collection: 8 assertions passed 9.38ms
- Fill collection: 15,003 assertions passed 558.79ms
  5000 rows inserted in 548.83ms i.e. 9110/s, aver. 109us, 3.1 MB/s
- Drop collection: no assertion 856us
- Fill collection bulk: 2 assertions passed 74.59ms
5000 rows inserted in 64.76ms i.e. 77204/s, aver. 12us, 7.2 MB/s
- Read collection: 30,003 assertions passed 2.75s
- Update collection: 7,503 assertions passed 784.26ms
- Delete some items: 4,002 assertions passed 370.57ms
1000 rows deleted in 96.76ms i.e. 10334/s, aver. 96us, 2.2 MB/s
Total failed: 0 / 56,527 - Direct with acknowledge PASSED 4.56s

1.2. Direct without acknowledge:
- Connect to local server: 6 assertions passed 1.30ms
- Drop and prepare collection: 8 assertions passed 8.59ms
- Fill collection: 15,003 assertions passed 192.59ms
  5000 rows inserted in 168.50ms i.e. 29673/s, aver. 33us, 4.4 MB/s
- Drop collection: no assertion 845us
- Fill collection bulk: 2 assertions passed 68.54ms
  5000 rows inserted in 58.67ms i.e. 85215/s, aver. 11us, 7.9 MB/s
- Read collection: 30,003 assertions passed 2.75s
  5000 rows read at once in 9.99ms i.e. 500150/s, aver. 1us, 38.5 MB/s
- Update collection: 7,503 assertions passed 446.48ms
  5000 rows updated in 96.27ms i.e. 51933/s, aver. 19us, 7.7 MB/s
- Delete some items: 4,002 assertions passed 297.26ms
  1000 rows deleted in 19.16ms i.e. 52186/s, aver. 19us, 2.8 MB/s
Total failed: 0 / 56,527 - Direct without acknowledge PASSED 3.77s

As you can see, the reading speed is not affected by the Write Concern settings.
But data writing can be multiple times faster, when each write command is not acknowledged.

Since there is no error handling, wcUnacknowledged is not to be used on production. You may use it for replication, or for data consolidation, e.g. feeding a database with a lot of existing data as fast as possible.
9.2. MongoDB + ORM = ODM

The mORMotMongoDB.pas unit is able to let any TSQLRecord class be persisted on a remote MongoDB server.

As a result, our ORM is able to be used as a NoSQL and Object-Document Mapping (ODM) (page 96) framework, with almost no code change. Any MongoDB database can be accessed via RESTful commands, using JSON over HTTP - see below (page 295).

This integration benefits from the other parts of the framework (e.g. our UTF-8 dedicated process, which is also the native encoding for BSON), so you can easily mix SQL and NoSQL databases with the exact same code, and are still able to tune any SQL or MongoDB request in your code, if necessary.

From the client point of view, there is no difference between a ORM or an ODM: you may use a SQL engine as a storage for ODM - via Shared nothing architecture (or sharding) (page 155) - or even a NoSQL database as a regular ORM, with denormalization (even if it may void most advantages of NoSQL).

9.2.1. Define the TSQLRecord class

In the database model, we define a TSQLRecord class, as usual:

```delphi
TSQLORM = class(TSQLRecord)
private
    fAge: integer;
    fName: RawUTF8;
    fDate: TDateTime;
    fValue: variant;
    fInts: TIntegerDynArray;
    fCreateTime: TCreateTime;
    fData: TSQLRawBlob;
published
    property Name: RawUTF8 read fName write fName stored AS_UNIQUE;
    property Age: integer read fAge write fAge;
    property Date: TDateTime read fDate write fDate;
    property Value: variant read fValue write fValue;
    property Ints: TIntegerDynArray index 1 read fInts write fInts;
    property Data: TSQLRawBlob read fData write fData;
    property CreateTime: TCreateTime read fCreateTime write fCreateTime;
end;
```

Note that we did not define any index ... values for the RawUTF8 property, as we need for external SQL databases, since MongoDB does not expect any restriction about text fields length (as far as I know, the only SQL engines which allow this natively without any performance penalty are SQLite3 and PostgreSQL).

The property values will be stored in the native MongoDB layout, i.e. with a better coverage than the SQL types recognized by our SynDB* unit:

<table>
<thead>
<tr>
<th>Delphi</th>
<th>MongoDB</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>int32</td>
<td></td>
</tr>
<tr>
<td>word</td>
<td>int32</td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>int32</td>
<td></td>
</tr>
<tr>
<td>cardinal</td>
<td>N/A</td>
<td>You should use Int64 instead</td>
</tr>
<tr>
<td>Field Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Int64</td>
<td>int64</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>enumeration</td>
<td>int32, store the ordinal value of the enumerated item (i.e. starting at 0 for the first element)</td>
<td></td>
</tr>
<tr>
<td>set</td>
<td>int64, each bit corresponding to an enumerated item (therefore a set of up to 64 elements can be stored in such a field)</td>
<td></td>
</tr>
<tr>
<td>single</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>extended</td>
<td>double, stored as double (precision lost)</td>
<td></td>
</tr>
<tr>
<td>currency</td>
<td>double, stored as double (MongoDB does not have a BSD type)</td>
<td></td>
</tr>
<tr>
<td>RawUTF8</td>
<td>UTF-8, this is the preferred field type for storing some textual content in the ORM</td>
<td></td>
</tr>
<tr>
<td>WinAnsiString</td>
<td>UTF-8, WinAnsi char-set (code page 1252) in Delphi</td>
<td></td>
</tr>
<tr>
<td>RawUnicode</td>
<td>UTF-8, UCS2 char-set in Delphi, as AnsiString</td>
<td></td>
</tr>
<tr>
<td>WideString</td>
<td>UTF-8, UCS2 char-set, as COM BSTR type (Unicode in all version of Delphi)</td>
<td></td>
</tr>
<tr>
<td>SynUnicode</td>
<td>UTF-8, Will be either WideString before Delphi 2009, or UnicodeString later</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>UTF-8, Not to be used before Delphi 2009 (unless you may loose some data during conversion) - RawUTF8 is preferred in all cases</td>
<td></td>
</tr>
<tr>
<td>TDateTime</td>
<td>datetime, ISO 8601 encoded date time</td>
<td></td>
</tr>
<tr>
<td>TDateTimeMS</td>
<td>datetime, ISO 8601 encoded date time</td>
<td></td>
</tr>
<tr>
<td>TTimeLog</td>
<td>int64, as proprietary fast Int64 date time</td>
<td></td>
</tr>
<tr>
<td>TModTime</td>
<td>int64, the server date time will be stored when a record is modified (as proprietary fast Int64)</td>
<td></td>
</tr>
<tr>
<td>TCreateTime</td>
<td>int64, the server date time will be stored when a record is created (as proprietary fast Int64)</td>
<td></td>
</tr>
<tr>
<td>TUnixTime</td>
<td>datetime, seconds since Unix epoch</td>
<td></td>
</tr>
<tr>
<td>TUnixMSTime</td>
<td>datetime, milliseconds since Unix epoch</td>
<td></td>
</tr>
<tr>
<td>TSQLRecord</td>
<td>int32, 32-bit RowID pointing to another record (warning: the field value contains pointer (RowID), not a valid object instance - the record content must be retrieved with late-binding via its ID using a PtrInt(Field) typecast or the Field.ID method), or by using e.g. CreateJoined() - is 64-bit on Win64</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Storage</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TID</td>
<td>int32/64</td>
<td>RowID pointing to another record - this kind of property is 64-bit compatible, so can handle values up to 9,223,720,036,854,775,808</td>
</tr>
<tr>
<td>TSQLRecordMany</td>
<td>nothing</td>
<td>data is stored in a separate pivot table; for MongoDB, you should better use data sharding, and an embedded sub-document</td>
</tr>
<tr>
<td>TRecordReferenceToBeDeleted</td>
<td>int32/64</td>
<td>store both ID and TSQLRecord type in a RecordRef-like value - with proper synchronization when the record is deleted</td>
</tr>
<tr>
<td>TPersistent</td>
<td>object</td>
<td>BSON object (from ObjectToJSON)</td>
</tr>
<tr>
<td>TCollection</td>
<td>array</td>
<td>BSON array of objects (from ObjectToJSON)</td>
</tr>
<tr>
<td>TObjectList</td>
<td>array</td>
<td>BSON array of objects (from ObjectToJSON) - see TJSONSerializer. RegisterClassForJSON below (page 309)</td>
</tr>
<tr>
<td>TStrings</td>
<td>array</td>
<td>BSON array of strings (from ObjectToJSON)</td>
</tr>
<tr>
<td>TRawUTF8List</td>
<td>array</td>
<td>BSON array of string (from ObjectToJSON)</td>
</tr>
<tr>
<td>any TObject</td>
<td>object</td>
<td>See TJSONSerializer. RegisterCustomSerializer below (page 306)</td>
</tr>
<tr>
<td>TSQLRawBlob</td>
<td>binary</td>
<td>This type is an alias to RawByteString - those properties are not retrieved by default: you need to use RetrieveBlobFields() or set ForceBlobTransfer / ForceBlobTransferTable[] properties</td>
</tr>
<tr>
<td>TByteDynArray</td>
<td>binary</td>
<td>Used to embed a BLOB property stored as BSON binary within a document - so that TSQLRawBlob may be restricted in the future to GridFS external content</td>
</tr>
<tr>
<td>dynamic arrays</td>
<td>array</td>
<td>if the dynamic array can be saved as true JSON, will be stored as BSON array - otherwise, will be stored in the TDynArray.SaveTo BSON binary format</td>
</tr>
<tr>
<td>variant</td>
<td>value</td>
<td>BSON number, text, date, object or array, depending on TDocVariant custom variant type (page 112) - or TBSONVariant stored value (e.g. to store native MongoDB types like ObjectID or Decimal128)</td>
</tr>
<tr>
<td>record</td>
<td>binary</td>
<td>BSON as defined in code by overriding TSQLRecord.InternalRegisterCustomProperties to produce true JSON</td>
</tr>
</tbody>
</table>

You can share the same TSQLRecord definition with MongoDB and other storage means, like external SQL databases. Unused information (like the index attribute) will just be ignored.

Note that TSQLRecord, TID and TRecordReference* published properties will automatically create an index on the corresponding field, and that a kind of ON DELETE SET DEFAULT tracking will take place for TSQLRecord and TRecordReference properties, and ON DELETE CASCADE for TRecordReferenceToBeDeleted - but not for TID, since we do not know which table to track.
9.2.2. Register the TSQLRecord class

On the server side (there won't be any difference for the client), you define a TMongoDBClient, and assign it to a given TSQLRecord class, via a call to StaticMongoDBRegister():

```pascal
    MongoClient := TMongoClient.Create('localhost', 27017);
    DB := MongoClient.Database['dbname'];
    Model := TSQLMongoModel.Create([TSQLORM]);
    Client := TSQLRestClientDB.Create(Model, nil, 'memory:', TSQLRestServerDB);
    if StaticMongoDBRegister(TSQLORM, fClient.Server, fDB, 'collectionname') = nil then
        raise Exception.Create('Error');
```

And... that's all!

If all the tables of a mORMot server should be hosted on a MongoDB server, you could call the StaticMongoDBRegisterAll() function instead:

```pascal
    StaticMongoDBRegisterAll(aServer, aMongoClient.Open(collectionname'));
```

If you want TSQLRecord.InitializeTable method to be called for void tables (and for instance create TSQLAuthGroup and TSQLAuthUser default content), you can execute the following command:

```pascal
    Client.Server.InitializeTables(INITIALIZETABLE_NOINDEX);
```

You can then execute any ORM command, as usual:

```pascal
    writeln(Client.TableRowCount(TSQLORM)=0);
```

As with external databases, you can specify the field names mapping between the objects and the MongoDB collection.

By default, the TSQLRecord.ID property is mapped to the MongoDB's _id field, and the ORM will populate this _id field with a sequence of integer values, just like any TSQLRecord table.

You can specify your own mapping, using e.g.:

```pascal
    aModel.Props[aClass].ExternalDB.MapField(..)
```

Since the field names are stored within the document itself, it may be a good idea to use shorter naming for the MongoDB collection. It may save some storage space, when working with a huge number of documents.

Once the TSQLRecord is mapped to a MongoDB collection, you can always have direct access to the corresponding TMongoCollection instance later on, using a simple transtyping:

```pascal
    (aServer.StaticDataServer[aClass] as TSQLRestStorageMongoDB).Collection
```

This may allow any specific task, including any tuned query or process.

9.2.3. ORM/ODM CRUD methods

You can add documents with the standard CRUD methods of the ORM, as usual:

```pascal
    R := TSQLORM.Create;
    try
        for i := 1 to COLL_COUNT do begin
            R.Name := 'Name ' + Int32ToUTF8(i);
            R.Age := i;
            R.Date := 1.0 * (30000+i);
            R.Value := _ObjFast(['num', i]);
            R.InTs := nil;
            R.DynArray(1).Add(i);
            assert(Client.Add(R, True) = i);
        end;
    finally
        R.Free;
    end;
```
As we already saw, the framework is able to handle any kind of properties, including complex types like *dynamic arrays* or variant.

In the above code, a TDocVariant document has been stored in R.Value, and a dynamic array of integer values is accessed via its index 1 shortcut and the TSQLRecord.DynArray() method.

The usual Retrieve / Delete / Update methods are available:

```pascal
R := TSQLORM.Create;
try
  for i := 1 to COLL_COUNT do begin
    Check(Client.Retrieve(i,R));
    // here R instance contains all values of one document, excluding BLOBs
  end;
finally
  R.Free;
end;
```

You can define a WHERE clause, as if the back-end where a regular SQL database:

```pascal
R := TSQLORM.CreateAndFillPrepare(Client,'ID=?',[i]);
try ...
```

### 9.2.4. ODM complex queries

To perform a query and retrieve the content of several documents, you can use regular CreateAndFillPrepare or FillPrepare methods:

```pascal
R := TSQLORM.CreateAndFillPrepare(Client,WHERE_CLAUSE,[WHERE_PARAMETERS]);
try
  n := 0;
  while R.FillOne do begin
    // here R instance contains all values of one document, excluding BLOBs
    inc(n);
  end;
  assert(n=COLL_COUNT);
finally
  R.Free;
end;
```

A WHERE clause can also be defined for CreateAndFillPrepare or FillPrepare methods. This WHERE clause could contain several expressions, joined with AND / OR.

Each of those expressions could use:

- The simple comparators = < <= <> > >=,
- An IN (....) clause,
- IS NULL / IS NOT NULL tests,
- A LIKE operation,
- Or even any ...DynArrayContains() specific function.

The *mORMot* ODM will convert this SQL-like statement into the optimized MongoDB query expression, using e.g. a regular expression for the LIKE operator.

The LIMIT, OFFSET and ORDER BY clauses will also be handled as expected. A special care should be taken for an ORDER BY on textual values: by design, *MongoDB* will always sort text with case-sensitivity, which is not what we expect: so our ODM will sort such content on client side, after having been retrieved from the *MongoDB* server. For numerical fields, *MongoDB* sorting features will be processed on the server side.

The COUNT(*) function will also be converted into the proper *MongoDB* API call, so that such operations will be as costless as possible. DISTINCT() MAX() MIN() SUM() AVG() functions and the
GROUP BY clause will also be converted into optimized MongoDB aggregation pipelines, on the fly. You could even set aliases for the columns (e.g. `max(RowID)` as `first`) and perform simple addition/substraction of an integer value.

Here are some typical WHERE clauses, and the corresponding MongoDB query document as generated by the ODM:

<table>
<thead>
<tr>
<th>WHERE clause</th>
<th>MongoDB Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Name=?', ['Name 43']</td>
<td><code>{Name: &quot;Name 43&quot;}</code></td>
</tr>
<tr>
<td>'Age&lt;?', [51]</td>
<td><code>{Age: {$lt: 51}}</code></td>
</tr>
<tr>
<td>'Age in (1,10,20)'</td>
<td><code>{Age: {$in: [1,10,20]}}</code></td>
</tr>
<tr>
<td>'Age in (1,10,20) and ID=?', [10]</td>
<td><code>{Age: {$in: [1,10,20]}, _id: 10}</code></td>
</tr>
<tr>
<td>'Age in (10,20) or ID=?', [30]</td>
<td><code>{or: [{Age: {$in: [10,20]}}, {id: 30}]}</code></td>
</tr>
<tr>
<td>'Name like ?', ['name 1/']</td>
<td><code>{Name: /^name 1/i}</code></td>
</tr>
<tr>
<td>'Name like ?', ['name 1']</td>
<td><code>{Name: /^name 1$/i}</code></td>
</tr>
<tr>
<td>'Name like ?', ['%ame 1%']</td>
<td><code>{Name: /ame 1/i}</code></td>
</tr>
<tr>
<td>'Data is null'</td>
<td><code>{Data: null}</code></td>
</tr>
<tr>
<td>'Data is not null'</td>
<td><code>{Data: {$ne: null}}</code></td>
</tr>
<tr>
<td>'Age&lt; limit 10', [51]</td>
<td><code>{Age: {$lt: 51}} + limit 10</code></td>
</tr>
<tr>
<td>'Age in (10,20) or ID=? order by ID desc', [30]</td>
<td><code>{$query: {or: [{Age: {$in: [10,20]}}, {id: 30}]}, orderby: {id: -1}}</code></td>
</tr>
<tr>
<td>'order by Name'</td>
<td><code>{}</code> + client side text sort by Name</td>
</tr>
<tr>
<td>'Age in (1,10,20) and IntegerDynArrayContains(Ints,?)', [10]</td>
<td><code>{Age: {$in: [1,10,20]}, Ints: {$in: [10]}}</code></td>
</tr>
<tr>
<td>Distinct(Age), <code>max(RowID)</code> as first, <code>count(Age)</code> as count group by age</td>
<td><code>{$group: {id: &quot;$Age&quot;, f1: {$max: &quot;$_id&quot;}, f2: {$sum: 1}}}, $project: {id: 0, &quot;Age&quot;: &quot;$_id&quot;, &quot;first&quot;: &quot;$f1&quot;, &quot;count&quot;: &quot;$f2&quot;}}</code></td>
</tr>
<tr>
<td><code>min(RowID), max(RowID), Count(RowID)</code></td>
<td><code>{$group: {id: null, f0: {$min: &quot;$_id&quot;}, f1: {$max: &quot;$_id&quot;}, f2: {$sum: 1}}}, $project: {id: 0, min(RowID): &quot;$f0&quot;, max(RowID): &quot;$f1&quot;, Count(RowID): &quot;$f2&quot;}}</code></td>
</tr>
<tr>
<td><code>min(RowID) as a, max(RowID)+1 as b, Count(RowID) as c</code></td>
<td><code>{$group: {id: null, f0: {$min: &quot;$_id&quot;}, f1: {$max: &quot;$_id&quot;}, f2: {$sum: 1}}}, $project: {id: 0, &quot;a&quot;: &quot;$f0&quot;, &quot;b&quot;: &quot;$f1&quot;, &quot;c&quot;: &quot;$f2&quot;}}</code></td>
</tr>
</tbody>
</table>

Note that parenthesis and mixed AND OR expressions are not handled yet. You could always execute any complex NoSQL query (e.g. using aggregation functions or the Map/Reduce pattern) by using directly the TMongoCollection methods.

But for most business code, mORMot allows to share the same exact code between your regular SQL databases or NoSQL engines. You do not need to learn the MongoDB query syntax: the ODM will compute the right expression for you, depending on the database engine it runs on.
9.2.5. BATCH mode

In addition to individual CRUD operations, our MongoDB is able to use BATCH mode for adding or deleting documents.

You can write the exact same code as with any SQL back-end:

```pascal
Client.BatchStart(TSQLORM);
R := TSQLORM.Create;
try
  for i := 1 to COLL_COUNT do begin
    R.Name := 'Name ' + Int32ToUTF8(i);
    R.Age := i;
    R.Date := 1.0*(30000+i);
    R.Value := _ObjFast(['num',i]);
    R.Ints := nil;
    R.DynArray(i).Add(i);
    assert(Client.BatchAdd(R,True)>=0);
  end;
finally
  R.Free;
end;
assert(Client.BatchSend(IDs)=HTTP_SUCCESS);
```

Or for deletion:

```pascal
Client.BatchStart(TSQLORM);
for i := 5 to COLL_COUNT do
  if i mod 5=0 then
    assert(fClient.BatchDelete(i)>=0);
assert(Client.BatchSend(IDs)=HTTP_SUCCESS);
```

Speed benefit may be huge in regard to individual Add/Delete operations, even on a local MongoDB server. We will see some benchmark numbers now.

9.2.6. ORM/ODM performance

You can take a look at Data access benchmark (page 199) to compare MongoDB as back-end for our ORM classes.

In respect to external SQL engines, it features very high speed, low CPU use, and almost no difference in use. We interfaced the BatchAdd() and BatchDelete() methods to benefit of MongoDB BULK process, and avoided most memory allocation during the process.

Here are some numbers, extracted from the MongoDBTests.dpr sample, which reflects the performance of our ORM/ODM, depending on the Write Concern mode used:

```
2. ORM

  2.1. ORM with acknowledge:
  - Connect to local server: 6 assertions passed 18.65ms
  - Insert: 5,002 assertions passed 521.25ms
    5000 rows inserted in 520.65ms i.e. 9603/s, aver. 104us, 2.9 MB/s
  - Insert in batch mode: 5,004 assertions passed 65.37ms
    5000 rows inserted in 65.07ms i.e. 76836/s, aver. 13us, 8.4 MB/s
  - Retrieve: 45,001 assertions passed 640.95ms
    5000 rows retrieved in 640.75ms i.e. 7803/s, aver. 128us, 2.1 MB/s
  - Retrieve all: 40,001 assertions passed 20.79ms
    5000 rows retrieved in 20.33ms i.e. 245941/s, aver. 4us, 27.1 MB/s
  - Retrieve one with where clause: 45,410 assertions passed 673.01ms
    5000 rows retrieved in 667.17ms i.e. 7494/s, aver. 133us, 2.0 MB/s
  - Update: 40,002 assertions passed 681.31ms
    5000 rows updated in 660.85ms i.e. 7565/s, aver. 132us, 2.4 MB/s
  - Blobs: 125,003 assertions passed 2.16s
```
5000 rows updated in 525.97ms i.e. 9506/s, aver. 105us, 2.4 MB/s
- Delete: 38,003 assertions passed 175.86ms
- Delete in batch mode: 33,003 assertions passed 34.71ms
1000 rows deleted in 14.90ms i.e. 67078/s, aver. 14us, 597 KB/s
Total failed: 0 / 376,435 - ORM with acknowledge PASSED 5.00s

2.2. ORM without acknowledge:
- Connect to local server: 6 assertions passed 16.83ms
- Insert: 5,002 assertions passed 179.79ms
5000 rows inserted in 179.15ms i.e. 27908/s, aver. 35us, 3.9 MB/s
- Insert in batch mode: 5,004 assertions passed 66.30ms
5000 rows inserted in 31.46ms i.e. 158891/s, aver. 6us, 17.5 MB/s
- Retrieve: 45,001 assertions passed 642.05ms
5000 rows retrieved in 641.85ms i.e. 7789/s, aver. 128us, 2.1 MB/s
- Retrieve all: 40,001 assertions passed 20.68ms
5000 rows retrieved in 20.26ms i.e. 246718/s, aver. 4us, 27.2 MB/s
- Update: 40,002 assertions passed 231.75ms
5000 rows updated in 765.24ms i.e. 7404/s, aver. 135us, 2.0 MB/s
- Blobs: 125,003 assertions passed 1.44s
5000 rows updated in 150.58ms i.e. 33202/s, aver. 8us, 3.6 MB/s
- Delete: 38,003 assertions passed 103.57ms
1000 rows deleted in 19.73ms i.e. 50668/s, aver. 19us, 2.4 MB/s
- Delete in batch mode: 33,003 assertions passed 47.50ms
1000 rows deleted in 364us i.e. 2747252/s, aver. 0us, 23.4 MB/s
Total failed: 0 / 376,435 - ORM without acknowledge PASSED 3.44s

As for direct MongoDB access, the wcUnacknowledged is not to be used on production, but may be very useful in some particular scenarios. As expected, the reading process is not impacted by the Write Concern mode set.
Before describing the Client-Server design of this framework, we may have to detail some standards it is based on:
- JSON as its internal data storage and transmission format;
- REST as its Client-Server architecture.

### 10.1. JSON

#### 10.1.1. Why use JSON?

As we just stated, the JSON format is used internally in this framework. By definition, the *JavaScript Object Notation* (JSON) is a standard, open and lightweight computer data interchange format.


<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Double precision floating-point format in <em>JavaScript</em>, generally depends on implementation. There is no specific integer type</td>
</tr>
<tr>
<td>String</td>
<td>Double-quoted Unicode, with backslash escaping</td>
</tr>
<tr>
<td>Boolean</td>
<td><code>true</code> or <code>false</code></td>
</tr>
<tr>
<td>Array</td>
<td>An ordered sequence of values, comma-separated and enclosed in square brackets; the values do not need to be of the same type</td>
</tr>
</tbody>
</table>
Object  An unordered collection of key:value pairs with the ': ' character separating the key and the value, comma-separated and enclosed in curly braces; the keys must be strings and should be distinct from each other

null  Empty/undefined value

Non-significant white space may be added freely around the "structural characters" (i.e. brackets "{ }", colons ": " and commas ", ").

The following example shows the JSON representation of an object that describes a person. The object has string fields for first name and last name, a number field for age, an object representing the person's address and an array of phone number objects.

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    },
    "phoneNumbers": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "fax",
            "number": "646 555-4567"
        }
    ]
}
```

Usage of this layout, instead of other like XML or any proprietary format, results in several particularities:
- Like XML, it is a text-based, human-readable format for representing simple data structures and associative arrays (called objects);
- It's easier to read (for both human beings and machines), quicker to implement, and much smaller in size than XML for most use;
- It's a very efficient format for data caching;
- Its layout allows to be rewritten in place into individual zero-terminated UTF-8 strings, with almost no wasted space: this feature is used for very fast JSON to text conversion of the tables results, with no memory allocation nor data copy;
- It's natively supported by the JavaScript language, making it a perfect serialization format in any AJAX (i.e. Web 2.0) or HTML5 Mobile application;
- The JSON format is simple, and specified in a short and clean RFC document;
- The default text encoding for both JSON and SQLite3 is UTF-8, which allows the full Unicode char-set to be stored and communicated;
- It is the default data format used by ASP.NET AJAX services created in Windows Communication Foundation (WCF) since .NET framework 3.5; so it's Microsoft officially "ready";
- For binary BLOB transmission, we simply encode the binary data as Base64; please note that, by default, BLOB fields are not transmitted over REST with other fields in JSON objects, see below (page 313) (only exception are dynamic array fields, which are transmitted within the other fields).

REST JSON serialization will indeed be used in our main ORM to process of any TSQLRecord published properties, and in the interface-based SOA architecture of the framework, for content transmission.
In the framework, the whole http://json.org... standard is implemented, with some exceptions/extensions:
- #0 characters will indicate the end of input, as with almost all JSON libraries - so if your text input contains a #0 char, please handle it as binary (note that other control chars are escaped as expected);
- You may use an "extended syntax" (used e.g. by MongoDB) by unquoting ASCII-only property names;
- Floating point numbers are sometimes limited to currency (i.e. 4 decimals), to ensure serialization/unserialization won't loose precision; but in such cases, it can be extended to the double precision via a set of options;
- There is no 53-bit limitation for integers, as with JavaScript: the framework handle 64-bit integer values - when using a JavaScript back-end, you may have to transmit huge values as text.

In practice, JSON has been found out to be very easy to work with and stable. A binary format is not used for transmission yet, but is available at other level of the framework, e.g. as an possible file format for in-memory TObjectList database engine (with our SynLZ compression - see Virtual Tables magic (page 225)).

10.1.2. Values serialization

Standard Delphi value types are serialized directly within the JSON content, in their textual representation. For instance, integer or Int64 are stored as numbers, and double values are stored as their corresponding floating-point representation.

All string content is serialized as standard JSON text field, i.e. nested with double quotes ("'). Since JSON uses UTF-8 encoding, it is one of the reasons why we introduced the RawUTF8 type, and use it everywhere in our framework.

10.1.3. Record serialization

In Delphi, the record has some nice advantages:
- record are value objects, i.e. accessed by value, not by reference - this can be very convenient, e.g. when defining Domain-Driven Design (page 99);
- record can contain any other record or dynamic array, so are very convenient to work with (no need to define sub-classes or lists);
- record variables can be allocated on stack, so won't solicited the global heap;
- record instances automatically freed by the compiler when they come out of scope, so you won't need to write any try..finally Free; end block.

Serialization of record values are therefore a must-have for a framework like mORMot. In practice, the record types should be defined as packed record, so that low-level access will be easier to manage by the serializers.

10.1.3.1. Automatic serialization via Enhanced RTTI

Since Delphi 2010, the compiler generates additional RTTI at compilation, so that all record fields are described, and available at runtime.

By the way, this enhanced RTTI is one of the reasons why executables did grow so much in newer versions of the compiler.

Our SynCommons.pas unit is able to use this enhanced information, and let any record be serialized via RecordLoad() and RecordSave() functions, and all internal JSON marshalling process.
In short, you have nothing to do. Just use your record as parameters, and, with Delphi 2010 and up, they will be serialized as valid JSON objects. The only restriction is that the records should be defined as packed record.

10.1.3.2. Serialization for older Delphi versions

Sadly, the information needed to serialize a record is available only since Delphi 2010.

If your application is developed on any older revision (e.g. Delphi 7, Delphi 2007 or Delphi 2009), you won't be able to automatically serialize records as plain JSON objects directly.

You have several paths available:
- By default, the record will be serialized as binary, and encoded as Base64 text;
- Or you can define method callbacks which will write or read the data as you expect;
- Or you can define the record layout as plain text.

Note that any custom serialization (either via callbacks, or via text definition), will override any previous registered method, even the mechanism using the enhanced RTTI. You can change the default serialization to easily meet your requirements. For instance, this is what SynCommons.pas does for any TGUID content, which is serialized as the standard JSON text layout (e.g. "C9A646D3-9C61-4CB7-BFCD-EE2522C8F633"), and not following the TGUID record layout as defined in the RTTI, i.e. {"D1":12345678,"D2":23023,"D3":9323,"D4":"0123456789ABCDEF"} - which is far from convenient.

10.1.3.2.1. Default Binary/Base64 serialization

On any version of the compiler prior to Delphi 2010, any record value will be serialized by default with a proprietary binary (and optimized) layout - i.e. via RecordLoad and RecordSave functions - then encoded as Base64, to be stored as plain text within the JSON stream.

A special UTF-8 prefix (which does not match any existing Unicode glyph) is added at the beginning of the resulting JSON string to identify this content as a BLOB, as such:

```
{ "MyRecord": "ï¿°w6nDoMOnYQ==" }
```

You will find in SynCommons.pas unit both BinToBase64 and Base64ToBin functions, very optimized for speed. Base64 encoding was chosen since it is standard, much more efficient than hexadecimal, and still JSON compatible without the need to escape its content.

When working with most part of the framework, you do not have anything to do: any record will by default follow this Base64 serialization, so you will be able e.g. to publish or consume interface-based services with records.

10.1.3.2.2. Custom serialization

Base64 encoding is pretty convenient for a computer (it is a compact and efficient format), but it is very limited about its interoperability. Our format is proprietary, and will use the internal Delphi serialization scheme: it means that it won't be readable nor writable outside the scope of your own mORMot applications. In a RESTful/SOA world, this sounds not like a feature, but a limitation.

Custom record JSON serialization can therefore be defined, as with any class - see below (page 306). It will allow writing and parsing record variables as regular JSON objects, ready to be consumed by any client or server. Internally, some callbacks will be used to perform the serialization.

In fact, there are two entry points to specify a custom JSON serialization for record:
When setting a custom *dynamic array* JSON serializer - see below (page 303) - the associated record will also use the same Reader and Writer callbacks;

- By setting explicitly serialization callbacks for the TypeInfo() of the record, with the very same TTextWriter. RegisterCustomJSONSerializer method used for dynamic arrays.

Then the Reader and Writer callbacks can be defined by two means:

- By hand, i.e. coding the methods with manual conversion to JSON text or parsing;
- Via some text-based type definition, which will follow the record layout, but will do all the marshalling (including memory allocation) on its own.

### 10.1.3.2.3. Defining callbacks

For instance, if you want to serialize the following record:

```
TSQLRestCacheEntryValue = record
  ID: TID;
  Timestamp: cardinal;
  JSON: RawUTF8;
end;
```

With the following code:

```
TTextWriter.RegisterCustomJSONSerializer(TypeInfo(TSQLRestCacheEntryValue),
  TTestServiceOrientedArchitecture.CustomReader,
  TTestServiceOrientedArchitecture.CustomWriter);
```

The expected format will be as such:
```
{"ID":1786554763,"Timestamp":323618765,"JSON":"D:\TestSQL3.exe"}
```

Therefore, the writer callback could be:

```
class procedure TTestServiceOrientedArchitecture.CustomWriter(  
  const aWriter: TTextWriter; const aValue);
var V: TSQLRestCacheEntryValue absolute aValue;
begin
  aWriter.AddJSONEscape(['ID',V.ID, 'Timestamp',Int64(V.Timestamp), 'JSON',V.JSON]);
end;
```

In the above code, the cardinal field named Timestamp is type-casted to an Int64: in fact, as stated by the documentation of the AddJSONEscape method, an array of const will handle by default any cardinal as an integer value (this is a limitation of the *Delphi* compiler). By forcing the type to be an Int64, the expected cardinal value will be transmitted, and not a wrongly negative versions for numbers > $7fffffff.

On the other side, the corresponding reader callback will be like:

```
class function TTestServiceOrientedArchitecture.CustomReader(P: PUTF8Char;  
  var aValue: TValue; out aValid: Boolean; aCustomVariantOptions: PDocVariantOptions): PUTF8Char;
begin
  result := JSONDecode(P,['ID','Timestamp','JSON'],@Values);
  if result=nil then
    aValid := false else begin
    V.ID := GetInt64(Values[0].Value);
    V.Timestamp := GetCardinal(Values[1].Value);
    Values[2].ToUTF8(V.JSON);
    aValid := true;
  end;
end;
```

Here JSONDecode() is used for fast deserialization of a JSON object.
10.1.3.2.4. Text-based definition

Writing those callbacks by hand could be error-prone, especially for the Reader event.

You can use the TTextWriter.RegisterCustomJSONSerializerFromText method to define the record layout in a convenient text-based format. Once more, those types need to be defined as packed record, so that the text layout definition will not depend on compiler-specific field alignment.

The very same TSQLRestCacheEntryValue can be defined as with a typical pascal record:

```pascal
const
__TSQLRestCacheEntryValue = 'ID: Int64; Timestamp: cardinal; JSON: RawUTF8';
```

Or with a shorter syntax:

```pascal
const
__TSQLRestCacheEntryValue = 'ID Int64 Timestamp cardinal JSON RawUTF8';
```

Both declarations will do the same definition. Note that the supplied text should match exactly the original record type definition: do not swap or forget any property!

By convention, we use two underscore characters (__) before the record type name, to easily identify the layout definition. It may indeed be convenient to write it as a constant, close to the record type itself, and not in-lined at RegisterCustomJSONSerializerFromText() call level.

Then you register your type as such:

```pascal
TTextWriter.RegisterCustomJSONSerializerFromText(
    TypeInfo(TSQLRestCacheEntryValue), __TSQLRestCacheEntryValue);
```

Now you are able to serialize any record value directly:

```pascal
Cache.ID := 10;
Cache.Timestamp := 200;
Cache.JSON := 'test';
U := RecordSaveJSON(Cache, TypeInfo(TSQLRestCacheEntryValue));
Check(U='{"ID":10,"Timestamp":200,"JSON":"test"}');
```

You can also unserialize some existing JSON content:

```pascal
U := '{"ID":210,"Timestamp":2200,"JSON":"test2"}';
RecordLoadJSON(Cache, @U[1], TypeInfo(TSQLRestCacheEntryValue));
Check(Cache.ID=210);
Check(Cache.Timestamp=2200);
Check(Cache.JSON='test2');
```

Note that this text-based definition is very powerful, and is able to handle any level of nested record or dynamic arrays.

By default, it will write the JSON content in a compact form, and will expect only existing fields to be available in the incoming JSON. You can specify some options at registration, to ignore all non defined fields. It can be very useful when you want to consume some remote service, and are interested only in a few fields.

For instance, we may define a client access to a RESTful service like api.github.com:

```pascal
type
TTestCustomJSONGitHub = packed record
  name: RawUTF8;
  id: cardinal;
  description: RawUTF8;
  fork: boolean;
  owner: record
    login: RawUTF8;
    id: cardinal;
end;
```
end;
TTestCustomJSONGitHubs = array of TTestCustomJSONGitHub;

const
__TTestCustomJSONGitHub = 'name RawUTF8 id cardinal description RawUTF8 '+
'fork boolean owner{login RawUTF8 id cardinal}';

Note the { } format to define a nested record, as a shorter alternative to a nested record .. end syntax.

It is also mandatory that you declare the record as packed. Otherwise, you may have unexpected access violation issues, since alignment may vary, depending on local setting, and compiler revision.

Now we can register the record layout, and provide some additional options:

TTextWriter.RegisterCustomJSONSerializerFromText(TypeInfo(TTestCustomJSONGitHub),
__TTestCustomJSONGitHub).Options := [soReadIgnoreUnknownFields,soWriteHumanReadable];

Here, we defined:
- soReadIgnoreUnknownFields to ignore any non defined field in the incoming JSON;
- soWriteHumanReadable to let the output JSON be more readable.

Then the JSON can be parsed then emitted as such:

var git: TTestCustomJSONGitHubs;
...
U := zendframeworkJson;
Check(DynArrayLoadJSON(git,@U[1],TypeInfo(TTestCustomJSONGitHubs))<>nil);
U := DynArraySaveJSON(git,TypeInfo(TTestCustomJSONGitHubs));

You can see that the record serialization is auto-magically available at dynamic array level, which is pretty convenient in our case, since the api.github.com RESTful service returns a JSON array.

It will convert 160 KB of very verbose JSON information:

```

Into the much smaller (6 KB) and readable JSON content, containing only the information we need:
During the parsing process, all unneeded JSON members will just be ignored. The parser will jump the data, without doing any temporary memory allocation. This is a huge difference with other existing Delphi JSON parsers, which first create a tree of all JSON values into memory, then allow to browse all the branches on request.

Note also that the fields have been ordered following the TTestCustomJSONGitHub record definition, which may not match the original JSON layout (here name/id fields order is inverted, and owner is set at the end of each item, for instance).

With mORMot, you can then access directly the content from your Delphi code as such:

```delphi
if git[0].id=8079771 then begin
  Check(git[0].name='Component_ZendAuthentication');
  Check(git[0].description='Authentication component from Zend Framework 2');
  Check(git[0].fork=true);
  Check(git[0].owner.login='zendframework');
  Check(git[0].owner.id=296074);
end;
```

Note that we do not need to use intermediate objects (e.g. via some obfuscated expressions like `gitarray.Value[0].Value['owner'].Value['login']`). Your code will be much more readable, will complain at compilation if you misspell any field name, and will be easy to debug within the IDE (since the record layout can be easily inspected).

The serialization is able to handle any kind of nested record or dynamic arrays, including dynamic arrays of simple types (e.g. array of integer or array of RawUTF8), or dynamic arrays of record:

```delphi
type
  TTestCustomJSONRecord = packed record
    A,B,C: integer;
    D: RawUTF8;
    E: record E1,E2: double; end;
    F: TDateTime;
  end;

  TTestCustomJSONArray = packed record
    A,B,C: integer;
    D: RawByteString;
    E: array of record E1: double; E2: string; end;
    F: TDateTime;
  end;

  TTestCustomJSONArraySimple = packed record
    A,B: Int64;
    C: array of SynUnicode;
    D: RawUTF8;
  end;
```

The corresponding text definitions may be:

```delphi
const
__TTestCustomJSONRecord = 'A,B,C integer D RawUTF8 E{E1,E2 double} F TDateTime';
__TTestCustomJSONArray = 'A,B,C integer D RawByteString E[E1 double E2 string] F TDateTime';
__TTestCustomJSONArraySimple = 'A,B Int64 C array of synunicode D RawUTF8';
```

The following types are handled by this feature:

<table>
<thead>
<tr>
<th>Delphi type</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>Serialized as JSON boolean</td>
</tr>
<tr>
<td>Data Type</td>
<td>Serialization Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>byte</td>
<td>Serialized as JSON number</td>
</tr>
<tr>
<td>word</td>
<td>Cardinal Int64</td>
</tr>
<tr>
<td>integer</td>
<td>Single currency TUnixTime</td>
</tr>
<tr>
<td>double</td>
<td>Serialized as JSON number</td>
</tr>
<tr>
<td>currency</td>
<td>TUnixTime</td>
</tr>
<tr>
<td>string</td>
<td>RawUTF8 SynUnicode WideString</td>
</tr>
<tr>
<td>DateTime</td>
<td>TTimeLog</td>
</tr>
<tr>
<td>RawByteString</td>
<td>Serialized as JSON null or Base64-encoded JSON string</td>
</tr>
<tr>
<td>RawJSON</td>
<td>Stored as un-serialized raw JSON content (e.g. any value, object or array)</td>
</tr>
<tr>
<td>TGUID</td>
<td>GUID serialized as JSON text</td>
</tr>
<tr>
<td>nested record</td>
<td>Serialized as JSON object</td>
</tr>
<tr>
<td></td>
<td>Identified as record ... end; or { ... } with its nested definition</td>
</tr>
<tr>
<td>nested registered record</td>
<td>Serialized as JSON corresponding the defined callbacks</td>
</tr>
<tr>
<td>dynamic array of record</td>
<td>Serialized as JSON array</td>
</tr>
<tr>
<td></td>
<td>Identified as array of ... or [ ... ]</td>
</tr>
<tr>
<td>dynamic array of simple types</td>
<td>Serialized as JSON array</td>
</tr>
<tr>
<td></td>
<td>Identified e.g. as array of integer</td>
</tr>
<tr>
<td>static array</td>
<td>Serialized as JSON array</td>
</tr>
<tr>
<td></td>
<td>Handled with enhanced RTTI, not via text definition yet</td>
</tr>
<tr>
<td>variant</td>
<td>Serialized as JSON, with full support of TDocVariant custom variant type (page 112)</td>
</tr>
</tbody>
</table>

For other types (like enumerations or sets), you can simply use the unsigned integer types corresponding to the binary value, e.g. byte word cardinal Int64 (depending on the sizeof() of the initial value).

For instance, void TTestCustomJSONRecord may be serialized as:

```json
{"A":0,"B":0,"C":0,"D":"","E":{"E1":0,"E2":0},"F":""}
```

Or void TTestCustomJSONArray may be serialized as:

```json
{"A":0,"B":0,"C":0,"D":null,"E":[""],"F":""}
```

Or void TTestCustomJSONArraySimple may be serialized as:

```json
{"A":0,"B":0,"C":[""],"D":["""]}
```

You can refer to the supplied regression tests (in TTestLowLevelTypes.EncodeDecodeJSON) for some more examples of custom JSON serialization.

### 10.1.4. Dynamic array serialization

#### 10.1.4.1. Standard JSON arrays

Note that dynamic arrays are handled in two separated contexts:

- Within the ORM part of the framework, they are stored as BLOB and always transmitted after
Base64 encoding - see TSQLRecord fields definition (page 131);

- Within the scope of interface-based services, dynamic arrays values and parameters are using the advanced JSON serialization made available in the TDynArray wrapper, i.e. could be either a true JSON array, or, in default, use generic binary and Base64 encoding, prior to Delphi 2010.

In fact, this TDynArray wrapper - see TDynArray dynamic array wrapper (page 107) - recognizes most common kind of dynamic arrays, like array of byte, word, integer, cardinal, Int64, double, currency, RawUTF8, SynUnicode, WinAnsiString, string. They will be serialized as a valid JSON array, i.e. a list of valid JSON elements of the matching type (number, floating-point value or string).

If you have any ideas of standard dynamic arrays which should be handled, feel free to post your proposal in the forum!

Since Delphi 2010, the framework will use the enhanced RTTI to create a JSON array corresponding to the data layout of each dynamic array item, just as for Record serialization (page 297).

For version of the compiler up to Delphi 2009, not-known dynamic arrays (like any array of packed record) will be serialized by default as binary, then Base64 encoded. This method will always work, but won't be easy to deal with from an AJAX client.

Of course, your applications can supply a custom JSON serialization for any other dynamic array, via the TTextWriter/RegisterCustomJSONSerializer() class method. Two callbacks are to be defined in association with dynamic array type information, in order to handle proper serialization and un-serialization of the JSON array.

As an alternative, you can call the RegisterCustomJSONSerializerFromText method to define the record layout in a convenient text-based format - see above.

In fact, if you register a dynamic array custom serializer, it will also be used for the associated internal record.

10.1.4.2. Customized serialization

As we already stated, it may be handy to change the default serialization.

For instance, we would like to serialize a dynamic array of the following record:

```delphi
tfv = packed record
  Major, Minor, Release, Build: integer;
  Main, Detailed: string;
end;

TFVs = array of TFV;
```

With the default serialization, such a dynamic array will be serialized either:

- As a Base64 encoded binary buffer, before Delphi 2010 - this won't be easy to understand from an AJAX client, for instance;
- As a JSON array of JSON object, with all property names listed within each object, since Delphi 2010 and its enhanced RTTI.

This default serialization can be overridden, by defining callbacks. It could be handy, e.g. if you do not like the fact that all field names are written in the data, which may be a waste of space:

```json
```

In order to add a custom serialization for this kind of record, we need to implement the two needed callbacks.

Our expected format will be a JSON array of all fields, i.e.:

```delphi
[1,2001,3001,4001,"1","1001"]
```
This layout is more than two times shorter than the default JSON object format.

We may have used another layout, e.g. using JSONEncode() function and a JSON object layout, or any other valid JSON content.

Here comes the writer:

```pascal
class procedure TCollTstDynArray.FVWriter(const aWriter: TTextWriter; const aValue);
var V: TFV absolute aValue;
begin
  aWriter.Add('[' V.Major,V.Minor,V.Release,V.Build,V.Main,V.Detailed],twJSONEscape);
end;
```

This event will write one entry of the dynamic array, without the last ',' (which will be appended by TTextWriter. AddDynArrayJSON). In this method, twJSOnEScape is used to escape the supplied string content as a valid JSON string (with double quotes and proper UTF-8 encoding).

Of course, the Writer is easier to code than the Reader itself:

```pascal
class function TCollTstDynArray.FVReader(P: PUTF8Char; var aValue; out aValid: Boolean): PUTF8Char;
var V: TFV absolute aValue;
  aValid := false;
  result := nil;
  if (P=nil) or (P^<>') then
    exit;
  inc(P);
  V.Major := GetNextItemCardinal(P);
  V.Minor := GetNextItemCardinal(P);
  V.Release := GetNextItemCardinal(P);
  V.Build := GetNextItemCardinal(P);
  V.Main := UTF8ToString(GetJSONField(P,P));
  V.Detailed := UTF8ToString(GetJSONField(P,P));
  if (P=nil) then
    exit;
  aValid := true;
  result := P; // ',' or ']' for last item of array
end;
```

The reader method shall return a pointer to the next separator of the JSON input buffer just after this item (either ',', or ']').

The registration process itself is as simple as:

```
TTextWriter.RegisterCustomJSONSerializer(TypeInfo(TFVs),
TCollTstDynArray.FVReader,TCollTstDynArray.FVWriter);
```

Then, from the user code point of view, this dynamic array handling won't change: once registered, the JSON serializers are used everywhere in the framework, as soon as this type is globally registered.

Here is a Writer method using a JSON object layout, which may be used for Delphi up to 2009, to obtain a serialization similar to the one generated via the enhanced RTTI.

```pascal
class procedure TCollTstDynArray.FVWriter2(const aWriter: TTextWriter; const aValue);
var V: TFV absolute aValue;
begin
  aWriter.AddJSONEscape(['Major',V.Major,'Minor',V.Minor,'Release',V.Release,
  'Build',V.Build,'Main',V.Main,'Detailed',V.Detailed]);
end;
```

This will create some JSON content as such:

```
```

We may also use similar callbacks, e.g. if we want the property names to be changed, or ignored
depending on some default values.

Then the corresponding Reader callback could be written as:

```pascal
class function TCollTstDynArray.FVReader2(P: PUTF8Char; var aValue; out aValid: Boolean): PUTF8Char;
var
V: TFV absolute aValue;
Values: array[0..5] of TValuePUTF8Char;
begin
aValid := false;
result := JSONDecode(P,['Major','Minor','Release','Build','Main','Detailed'],@Values);
if result=nil then
exit; // result^ = ',' or ']' for last item of array
V.Major := Values[0].ToInteger;
V.Minor := Values[1].ToInteger;
V.Release := Values[2].ToInteger;
V.Build := Values[3].ToInteger;
V.Main := Values[4].ToString;
V.Detailed := Values[5].ToString;
aValid := true;
end;

Most of the JSON decoding process is performed within the JSONDecode() function, which will let Values[].Value/ValueLen couples point to null-terminated un-escaped content within the P^ buffer. In fact, unserialization will do no memory allocation, and will therefore be very fast.

If you want to go back to the default binary + Base64 encoding serialization, you may run the registering method as such:

```pascal
TTextWriter.RegisterCustomJSONSerializer(TypeInfo(TFVs),nil,nil);
```

Or calling the text-based registration with a void definition:

```pascal
TTextWriter.RegisterCustomJSONSerializerFromText(TypeInfo(TTestCustomJSONGitHub),'');
```

You can define now your custom JSON serializers, starting for the above code as reference, or via the RegisterCustomJSONSerializerFromText() method text-based definition.

Note that if the record corresponding to its item dynamic array has some associated RTTI (i.e. if it contains some reference-counted types, like any string), it will be serialized as JSON during the mORMot service process, just as stated with Record serialization (page 297).

### 10.1.5. TSQLRecord TPersistent TStrings TRawUTF8List

Classes with published properties, i.e. every class inheriting from TPersistent or our ORM-dedicated TSQLRecord class will be serialized as a true JSON object, containing all their published properties values. See TSQLRecord fields definition (page 131) for a corresponding table with the ORM database types and the JSON content.

List of Delphi strings, i.e. TStrings kind of classes will be serialized as a JSON array of strings. This is the reason why we also introduced a dedicated TRawUTF8List class, for direct UTF-8 content storage, via our dedicated RawUTF8 type, reducing the need of encoding conversion, therefore increasing process speed.

### 10.1.6. TObject serialization

In fact, any TObject can be serialized as JSON in the whole framework: not only for the ORM part (for published properties), but also for SOA (as parameters of interface-based service methods). All JSON serialization is centralized in ObjectToJSON() and JOSEmToObject() (aka TJSONSerializer.WriteObject) functions.
10.1.6.1. Custom class serialization

In some cases, it may be handy to have a custom serialization, for instance if you want to manage some third-party classes, or to adapt the serialization scheme to a particular purpose, at runtime.

You can add a customized serialization of any class, by calling the TJSONSerializer.RegisterCustomSerializer class method. Two callbacks are to be defined for a specific class type, and will be used to serialize or un-serialize the object instance. The callbacks are class methods (procedure() of object), and not plain functions (for some evolved objects, it may have sense to use a context during serialization).

In the current implementation of this feature, callbacks expect low-level implementation. That is, their implementation code shall follow function TJSONObject() patterns, i.e. calling low-level GetJSONObject() function to decode the JSON content, and follow function TJSONSerializer.WriteObject() patterns, i.e. aSerializer.Add/AddInstanceName/AddJSONEscapeString to encode the class instance as JSON.

Note that the process is called outside the "{...}" JSON object layout, allowing any serialization scheme: even a class content can be serialized as a JSON string, JSON array or JSON number, on request.

For instance, we'd like to customize the serialization of this class (defined in SynCommons.pas):

```delphi
TFileVersion = class
  protected
    fDetailed: string;
    fBuildDateTime: TDateTime;
  public
    Major: Integer;
    Minor: Integer;
    Release: Integer;
    Build: Integer;
    BuildYear: integer;
    Main: string;
  published
    property Detailed: string read fDetailed write fDetailed;
    property BuildDateTime: TDateTime read fBuildDateTime write fBuildDateTime;
end;
```

By default, since it has been defined within {$M+} ... {$M-} conditionals, RTTI is available for the published properties (just as if it were inheriting from TPersistent). That is, the default JSON serialization will be for instance:

```
{"Detailed":"1.2.3.4","BuildDateTime":"1911-03-14T00:00:00"}
```

This is what is expected when serialized within a TSynLog content - see below (page 631) - or for current AJAX use.

We would like to serialize this class as such:

```
{"Major":1,"Minor":2001,"Release":3001,"Build":4001,"Main":"1","BuildDateTime":"1911-03-14"}
```

We will therefore define the Writer callback, as such:

```delphi
class procedure TCollTstDynArray.FVClAssWriter(const aSerializer: TJSONSerializer; aValue: TObject; aOptions: TTextWriterWriteObjectOptions);
var V: TFileVersion absolute aValue;
begin
  aSerializer.AddJSONEscape(['Major',V.Major,'Minor',V.Minor,'Release',V.Release,
                             'Build',V.Build,'Main',V.Main,'BuildDateTime',DateTimeToIso8601Text(V.BuildDateTime)]);
end;
```

Most of the JSON serialization work will be made within the AddJSONEscape method, expecting the
JSON object description as an array of name/value pairs.

Then the associated Reader callback could be, for instance:

```pascal
class function TCollTstDynArray.FVClassReader(const aValue: TObject; aFrom: PUTF8Char; var aValid: Boolean; aOptions: TJSONObjectToObjectOptions): PUTF8Char;
var
  V: TFileVersion absolute aValue;
  Values: array[0..5] of TValuePUTF8Char;
begin
  result := JSONDecode(aFrom,['Major','Minor','Release','Build','Main','BuildDateTime'],@Values);
  aValid := (result<>nil);
  if aValid then begin
    V.Major := Values[0].ToInteger;
    V.Minor := Values[1].ToInteger;
    V.Release := Values[2].ToInteger;
    V.Build := Values[3].ToInteger;
    V.Main := Values[4].ToString;
    V.BuildDateTime := Iso8601ToDateTimePUTF8Char(Values[5].Value,Values[5].ValueLen);
  end;
end;
```

Here, the JSONDecode function will un-serialize the JSON object into an array of PUTF8Char values, without any memory allocation (in fact, Values[].Value will point to un-escaped and #0 terminated content within the aFrom memory buffer. So decoding is very fast.

Then, the registration step will be defined as such:

```pascal
TJSONSerializer.RegisterCustomSerializer(TFileVersion,
TCollTstDynArray.FVClassReader,TCollTstDynArray.FVClassWriter);
```

If you want to disable the custom serialization, you may call the same method as such:

```pascal
TJSONSerializer.RegisterCustomSerializer(TFileVersion,nil,nil);
```

This will reset the JSON serialization of the specified class to the default serializer (i.e. writing of published properties).

The above code uses some low-level functions of the framework (i.e. AddJSONEscape and JSONDecode) to implement serialization as a JSON object, but you may use any other serialization scheme, on need. That is, you may serialize the whole class instance just as one JSON string or numerical value, or even a JSON array. It will depend of the implementation of the Reader and Writer registered callbacks.

10.1.6.2. Custom field names serialization

If your customization just expect changing some property names, you may use TJSONSerializer.RegisterCustomSerializerFieldNames class method.

For instance, given the following class:

```pascal
type
  TMyClass = class(TSynPersistent)
private
  FLength: Integer;
  FColor: Integer;
  FName: RawUTF8;
published
  property Color: Integer read FColor write FColor;
  property Length: Integer read FLength write FLength;
  property Name: RawUTF8 read FName write FName;
end;
```

You may use default serialization as such:

```pascal
var
```

SAD - mORMot Framework - Rev. 1.18

Page 309 of 2552
0: TMyClass;
json: RawUTF8;
begin
  0 := TMyClass.Create;
  0.Color := 10;
  0.Length := 20;
  0.Name := 'one';
  json := ObjectToJSON(0);
  writeln(json);  // {"Color":10,"Length":20,"Name":"one"}
end

Then switch to customized serialization:

TJSONSerializer.RegisterCustomSerializerFieldNames(TMyClass, ['name','length'], ['n','len']);
json := ObjectToJSON(0);
writeln(json);  // {"Color":10,"len":20,"n":"one"}

And back to normal/default serialization:

TJSONSerializer.RegisterCustomSerializerFieldNames(TMyClass, [], []);
json := ObjectToJSON(0);
writeln(json);  // {"Color":10,"Length":20,"Name":"one"}

You could ignore some fields, by setting the destination name to '':

TJSONSerializer.RegisterCustomSerializerFieldNames(TMyClass, ['length'], ['']);
json := ObjectToJSON(0);
writeln(json);  // {"Color":10,"Name":"one"}
0.Free;

This method may therefore help working with pre-existing JSON objects, for instance retrieved from a third-party REST server.

Note that the TJSONSerializer.RegisterCustomSerializerFieldNames method won't accept TSQLRecord classes, since ORM serialization is handled in its own (optimized) set - and you could use ORM-level mapping if needed - see Database-first ORM (page 270).

10.1.6.3. TObjectList serialization

You can even serialize TObjectList instances as a valid JSON array, with the ability to store each instance class name, so allowing the storage of non uniformous lists of objects.

Calling TJSONSerializer registrado() is just needed to register each TObject class in its internal tables, and be able to create instances from a class name serialized in each JSON object.

In fact, if ObjectToJSON() or TJSONWriter.WriteObject() have their woStoreClassName option defined, a new "ClassName": field will be written as first field of the serialized JSON object.

This new "ClassName" field will be recognized:
- by JSONToObject() for TObjectList members,
- and by the new JSONToNewObject() method.

Note that all TSQLRecord classes of a model are automatically registered via a call to TJSONSerializer registrado(): you do not have to register them, and can directly serialize TObjectList of TSQLRecords.

As a consequence, this kind of code can now work:

// register the type (but Classes.RegisterClass list is also checked)
TJSONSerializer.registrado(TComplexNumber);
// create an instance by reading the textual class name field
J := '{"ClassName":"TComplexNumber", "Real": 10.3, "Imaginary": 7.92 }';
P := @J[1];  // make local copy of constant
Comp := TComplexNumber(JSONToNewObject(P,Valid));
// here Comp is a valid unserialized object :)

Check(Valid);
    Check(Comp.ClassType=TComplexNumber);
    CheckSame(Comp.Real,10.3);
    CheckSame(Comp.Imaginary,7.92);
    // do not forget to free the memory (Comp can be nil if JSON was not valid)
    Comp.Free;

Internal TObjectList process will therefore rely on a similar process, creating the proper class instances on the fly. You can even have several classes appearing in one TObjectList: the only prerequisite is that all class types shall have been previously registered on both sides, by a call to TJSONSerializer/RegisterClassForJSON().
10.2. REST

10.2.1. What is REST?

*Representational state transfer* (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web. As such, it is not just a method for building "web services". The terms "representational state transfer" and "REST" were introduced in 2000 in the doctoral dissertation of Roy Fielding, one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification, on which the whole Internet rely.

There are 5 basic fundamentals of web which are leveraged to create REST services:
- Everything is a Resource;
- Every Resource is Identified by a Unique Identifier;
- Use Simple and Uniform Interfaces;
- Communication is Done by Representation;
- Every Request is Stateless.

10.2.1.1. Resource-based

Internet is all about getting data. This data can be in a format of web page, image, video, file, etc. It can also be a dynamic output like get customers who are newly subscribed. The first important point in REST is start thinking in terms of resources rather than physical files.

You access the resources via some URI, e.g.
- http://www.mysite.com/Customer/1001 - Dynamic Resource returning XML or JSON content;

10.2.1.2. Unique Identifier

Older web techniques, e.g. *aspx* or *ColdFusion*, did request a resource by specifying parameters, e.g.

http://www.mysite.com/Default.aspx?a=1;a=2&b=1&a=3

In REST, we add one more constraint to the current URI: in fact, every URI should uniquely represent every item of the data collection.

For instance, you can see the below unique URI format for customer and orders fetched:

<table>
<thead>
<tr>
<th>Customer data</th>
<th>URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Customer details with name &quot;dupont&quot;</td>
<td><a href="http://www.mysite.com/Customer/dupont">http://www.mysite.com/Customer/dupont</a></td>
</tr>
<tr>
<td>Get Customer details with name &quot;smith&quot;</td>
<td><a href="http://www.mysite.com/Customer/smith">http://www.mysite.com/Customer/smith</a></td>
</tr>
<tr>
<td>Get orders placed by customer &quot;dupont&quot;</td>
<td><a href="http://www.mysite.com/Customer/dupont/Orders">http://www.mysite.com/Customer/dupont/Orders</a></td>
</tr>
<tr>
<td>Get orders placed by customer &quot;smith&quot;</td>
<td><a href="http://www.mysite.com/Customer/smith/Orders">http://www.mysite.com/Customer/smith/Orders</a></td>
</tr>
</tbody>
</table>

Here, "dupont" and "smith" are used as unique identifiers to specify a customer. In practice, a name is far from unique, therefor most systems use an unique ID (like an integer, a hexadecimal number or a GUID).
10.2.1.3. Interfaces

To access those identified resources, basic CRUD activity is identified by a set of HTTP verbs:

<table>
<thead>
<tr>
<th>HTTP method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>List the members of the collection (one or several)</td>
</tr>
<tr>
<td>PUT</td>
<td>Update a member of the collection</td>
</tr>
<tr>
<td>POST</td>
<td>Create a new entry in the collection</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete a member of the collection</td>
</tr>
</tbody>
</table>

Then, at URI level, you can define the type of collection, e.g. http://www.mysite.com/Customer to identify the customers or http://www.mysite.com/Customer/1234/Orders to access a given order.

This combination of HTTP method and URI replace a list of English-based methods, like GetCustomer / InsertCustomer / UpdateOrder / RemoveOrder.

10.2.1.4. By Representation

What you are sending over the wire is in fact a representation of the actual resource data.

The main representation schemes are XML and JSON.

For instance, here is how a customer data is retrieved from a GET method:

```xml
<Customer>
  <ID>1234</ID>
  <Name>Dupond</Name>
  <Address>Tree street</Address>
</Customer>
```

Below is a simple JSON snippet for creating a new customer record with name and address (since we create a new record, here we named him "Dupond" - with an ending D - not "Dupont"):

```json
{"Customer": {"Name":"Dupond", "Address":"Tree street"}}
```

As a result to this data transmitted with a POST command, the RESTful server will return the just-created ID.

See JSON (page 295) for the reasons why in mORMot, we prefer to use JSON format.

10.2.1.5. Stateless

Every request should be an independent request so that we can scale up using load balancing techniques.

Independent request means with the data also send the state of the request so that the server can carry forward the same from that level to the next level.

See below (page 314) for more details.
10.2.2. RESTful mORMot

The *Synops e mORMot Framework* was designed in accordance with Fielding's REST architectural style without using HTTP and without interacting with the World Wide Web. Such Systems which follow REST principles are often referred to as "RESTful". Optionally, the Framework is able to serve standard HTTP/1.1 pages over the Internet (by using the mORMotHttpClient / mORMotHttpServer units and the TSQLHttpClient and TSQLHttpServer classes), in an embedded low resource and fast HTTP server.

The standard RESTful methods are implemented, i.e. GET/PUT/POST/DELETE.

The following methods were added to the standard REST definition, for locking individual records and for handling database transactions (which speed up database process):

- LOCK to lock a member of the collection;
- UNLOCK to unlock a member of the collection;
- BEGIN to initiate a transaction;
- END to commit a transaction;
- ABORT to rollback a transaction.

The GET method has an optional pagination feature, compatible with the YUI DataSource Request Syntax for data pagination - see TSQLRestServer.URI method and http://developer.yahoo.com/yui/datatable/#data.. Of course, this breaks the "Every Resource is Identified by a Unique Identifier" RESTful principle - but it is much more easy to work with, e.g. to implement paging or custom filtering.

From the Delphi code point of view, a RESTful Client-Server architecture is implemented by inheriting some common methods and properties from a main class.

![TSQ LRestClient classes hierarchy](image)

This diagram states how the TSQLRest class implements a common ancestor for both Client and Server classes.

10.2.2.1. BLOB fields

BLOB fields are defined as TSQLRawBlob published properties in the classes definition - which is an alias to the RawByteString type (defined in SynCommons.pas for Delphi up to 2007, since it appeared only with Delphi 2009). But their content is not included in standard RESTful methods of the framework, to spare network bandwidth.

The RESTful protocol allows BLOB to be retrieved (GET) or saved (PUT) via a specific URL, like:

```
ModelRoot/TableName/TableID/BlobFieldName
```
This is even better than the standard JSON encoding, which works well but convert BLOB to/from hexadecimal values, therefore need twice the normal size of it. By using such dedicated URL, data can be transferred as full binary.

Some dedicated methods of the generic TSQLRest class handle BLOB fields: RetrieveBlob and UpdateBlob.

10.2.2.2. JSON representation

The "04 - HTTP Client-Server" sample application available in the framework source code tree can be used to show how the framework is AJAX-ready, and can be proudly compared to any other REST server (like CouchDB) also based on JSON.

First desactivate the authentication - see below (page 546) - by changing the parameter from true to false in Unit2.pas:

```pascal
DB := TSQLRestServerDB.Create(Model,ChangeFileExt(paramstr(0),'.db3'),
false);
```

and by commenting the following line in Project04Client.dpr:

```pascal
Form1.Database := TSQLHttpClient.Create(Server,'8080',Form1.Model);
// TSQLHttpClient(Form1.Database).SetUser('User','synopse');
Application.Run;
```

Then you can use your browser to test the JSON content:

- Start the Project04Server.exe program: the background HTTP server, together with its SQLite3 database engine;
- Start any Project04Client.exe instances, and add/find any entry, to populate the database a little;
- Close the Project04Client.exe programs, if you want;
- Open your browser, and type into the address bar:

  ```
  http://localhost:8080/root
  ```

  - You'll see an error message:

  ```
  TSQLHttpGetServer Server Error 400
  ```

  Type into the address bar:

  ```
  http://localhost:8080/root/SampleRecord
  ```

  - You'll see the result of all SampleRecord IDs, encoded as a JSON list, e.g.:

  ```
  [{"ID":1},{"ID":2},{"ID":3},{"ID":4}]
  ```

  Type into the address bar:

  ```
  http://localhost:8080/root/SampleRecord/1
  ```

  - You'll see the content of the SampleRecord of ID=1, encoded as JSON, e.g.:

  ```
  {"ID":1,"Time":"2010-02-08T11:07:09","Name":"AB","Question":"To be or not to be"}
  ```

  - Type into the address bar any other REST command, and the database will reply to your request...

You have got a full HTTP/SQLite3 RESTful JSON server in less than 400 KB.

Note that Internet Explorer or old versions of Firefox do not recognize the application/json; charset=UTF-8 content type to be viewed internally. This is a limitation of those softwares, so above requests will download the content as .json files, but won't prevent AJAX requests to work as expected.

10.2.2.3. Stateless ORM
Our framework is implementing REST as a stateless protocol, just as the HTTP/1.1 protocol it could use as its communication layer.

A stateless server is a server that treats each request as an independent transaction that is unrelated to any previous request.

At first, you could find it a bit disappointing from a classic Client-Server approach. In a stateless world, you are never sure that your Client data is up-to-date. The only place where the data is safe is the server. In the web world, it's not confusing. But if you are coming from a rich Client background, this may concern you: you should have the habit of writing some synchronization code from the server to replicate all changes to all its clients. This is not necessary in a stateless architecture any more.

The main rule of this architecture is to ensure that the Server is the only reference, and that the Client is able to retrieve any pending update from the Server side. That is, always modify a record content on a server side, then refresh the client to retrieve the modified value. Do not modify the client side directly, but always pass through the Server. The UI components of the framework follow these principles. Client-side modification could be performed, but must be made in a separated autonomous table/database. This will avoid any synchronization problem in case of concurrent client modification.
Synopse mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

10.3. REST and JSON
10.3.1. JSON format density
Most common RESTful JSON used a verbose format for the JSON content: see for example
http://bitworking.org/news/restful_json.. which proposed to put whole URI in the JSON content;
[
"http://example.org/coll/1",
"http://example.org/coll/2",
"http://example.org/coll/3",
...
"http://example.org/coll/N",
]

The REST implementation of the framework will return most concise JSON content, containing an
array of objects:
[{"ID":1},{"ID":2},{"ID":3},{"ID":4}]

Depending on a setting, mORMot servers may in fact returns this alternative (see below non expanded
format), which can be shorter, since it does not replicate field names:
{"fieldCount":1,"values":["ID",1,2,3,4,5,6,7]}

which preserves bandwidth and human readability: if you were able to send a GET request to the URI
http://example.org/coll you will be able to append this URI at the beginning of every future
request, doesn't it make sense?
In all cases, the Synopse mORMot Framework always returns the JSON content just as a pure response
of a SQL query, with an array and field names.

10.3.2. JSON (not) expanded layouts
Note that our JSON content has two layouts, which can be produced according to the
TSQLRestServer.NoAJAXJSON property:
1. the "expanded" or standard/AJAX layout, which allows you to create pure JavaScript objects from
the JSON content, because the field name / JavaScript object property name is supplied for every
value:
[{"ID":0,"Int":0,"Test":"abcde+¬ef+á+¬","Unicode":"abcde+¬ef+á+¬","Ansi":"abcde+¬ef+á+¬","ValFloat
":3.14159265300000E+0000,"ValWord":1203,"ValDate":"2009-03-10T21:19:36","Next":0},{..}]

2. the "not expanded" layout, which reflects exactly the layout of the SQL request: first line/row are
the field names, then all next lines.row are the field content:
{"fieldCount":9,"values":["ID","Int","Test","Unicode","Ansi","ValFloat","ValWord","ValDate","Next"
,0,0,"abcde+¬ef+á+¬","abcde+¬ef+á+¬","abcde+¬ef+á+¬",3.14159265300000E+0000,1203,"2009-03-10T21:19
:36",0,..]}

By

default, the NoAJAXJSON property is set to true when the TSQLRestServer.
ExportServerNamedPipe is called: if you use named pipes for communication, you probably won't
use a JavaScript client since all browsers communicate via HTTP only!
But otherwise, NoAJAXJSON property is set to false. You could force its value to true and you will
save some bandwidth if JavaScript is never executed: even the parsing of the JSON Content will be
faster with Delphi if JSON content is not expanded.
SAD - mORMot Framework - Rev. 1.18

Page 317 of 2552


Synopse mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

In this "not expanded" layout, the following JSON content:
[{"ID":1},{"ID":2},{"ID":3},{"ID":4},{"ID":5},{"ID":6},{"ID":7}]

will be transfered as shorter:
{"fieldCount":1,"values":["ID",1,2,3,4,5,6,7]}

10.3.3. JSON global cache
A global cache, at SQlite3 level, is used to enhance the framework scaling, featuring JSON storage for
its result encoding.
In order to speed-up the server response time, especially in a concurrent client access, the internal
database engine is not to be called on every request. In fact, a global cache has been introduced to
store in memory the latest SQL SELECT statements results, directly in JSON.
The SQLite3 engine access is protected at SQL/JSON cache level, via DB.LockJSON() calls in most
TSQLRestServerDB methods.
A TSynCache instance is instantiated within the TSQLDataBase internal global instance, with the
following line:
constructor TSQLRestServerDB.Create(aModel: TSQLModel; aDB: TSQLDataBase;
aHandleUserAuthentication: boolean);
begin
fStatementCache.Init(aDB.DB);
aDB.UseCache := true; // we better use caching in this JSON oriented use
(...)

This will enable a global JSON cache at the SQL level. This cache will be reset on every INSERT, UPDATE
or DELETE SQL statement, whatever the corresponding table is.
If you need to disable the JSON cache for a particular request, add the SQLDATABASE_NOCACHE text, i.e.
the '/*nocache*/' text comment, anywhere in the SQL statement, e.g. in the ORM WHERE clause. It
will indicate to TSQLDataBase to not cache the returned JSON content. It may be usefull e.g. if you
pass a pointer as PtrInt(aVariable) bound parameter, which may have the very same integer
reference value, but diverse content.
In practice, this global cache was found to be efficient, even if its implementation is some kind of
"naive". It is in fact much more tuned than other HTTP-level caching mechanisms used in most
client-server solutions (using e.g. a Squid proxy) - since our caching is at the SQL level, it is shared
among all CRUD / Restful queries, and is also indenpendent from the authentication scheme, which
pollutes the URI. Associated with the other levels of cache - see ORM Cache (page 174) - the
framework scaling was found to be very good.

SAD - mORMot Framework - Rev. 1.18

Page 318 of 2552


# 11. Client-Server process

Adopt a mORMot

## 11.1. Client-Server cheat sheet

Before deeping into the details, and presenting all the mORMot framework Client-Server abilities, let's step back, and look at the big picture.

In practice, for your project, you will have several possibilities to create a Client-Server system. ORM, SOA and MVC can all be accessed remotely, and it may not be easy to find out which method is preferred to implement, in the context of a production system.

<table>
<thead>
<tr>
<th>Method</th>
<th>Best for</th>
<th>Beware</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA Interfaces</td>
<td>RPC/REST</td>
<td>RPC</td>
</tr>
<tr>
<td>SOA Methods</td>
<td>Full REST/HTTP</td>
<td>Verbose</td>
</tr>
<tr>
<td>MVC Web</td>
<td>Web site + AJAX</td>
<td>HTML-oriented</td>
</tr>
<tr>
<td>ORM REST</td>
<td>Tests or internal use</td>
<td>Security/design flows</td>
</tr>
</tbody>
</table>

In a nutshell,

- **SOA Interfaces** - see below (page 419) - is the preferred way to build both public and private services: both client and server code will be defined from interface types, including sessions management, stubbing/mocking, documentation generation, and security features.
- **SOA Methods** - see below (page 373) - will open full access to REST/HTTP details of each request, so may be needed to conform to a more REST, less RPC implementation - but the client side will need to be written by hand, and the server side could be more verbose to implement.
- **MVC Web** - see below (page 519) - is the way to go if you expect to develop mostly dynamic web pages, and sometimes consume some JSON content from JavaScript if needed, by accessing its `url/json` sub path.
- **ORM REST** - see below (page 341) - exposes all data automatically, but should better not be used on production for public APIs for architecture and security reasons, since it is directly tied to the datastore. It could be exposed internally, or for debugging/testing.
- remember that *any combination of the four previous framework features* could be defined in the same TSQLRestServer instance, so you can just pickup what fits best your needs.

We will now present all those communication features, but you may focus on **SOA Interfaces**, and its associated samples, when implementing your project, and go back to other details of this exhaustive documentation, only if needed.

### 11.2. Protocols

The *mORMot* framework can be used either stand-alone, or in a Client-Server model, via several communication layers:

- Fast in-process access (an executable file using a common library, for instance);
- Windows Messages, only locally on the same computer, which are very fast for small content;
- Named pipes, which can be used locally between a Server running as a Windows service and some Client instances;
- HTTP/1.1 over TCP/IP, for remote access.

See *General mORMot architecture - Client / Server* (page 74) about this Client-Server architecture.

The framework allow you to create either a TSQLHttpClient, TSQLRestClientURIDll, TSQLRestClientURINamedPipe or TSQLRestClientURIErrorMessage instance to access to your data according to the communication protocol used for the server.

Abilities will depend on the protocol used. For instance, HTTP may sounds slower than alternatives, but it is the best protocol for remote access of concurrent clients, even running locally. For instance, *mORMot*’s http.sys based server is able to serve 50,000 concurrent connections without any problem, but you should better not attempt connecting more than a dozen clients via named pipes or messages...
Here are some general information about available communication layers:

<table>
<thead>
<tr>
<th></th>
<th>In-process</th>
<th>Windows Messages</th>
<th>Named pipes</th>
<th>HTTP</th>
</tr>
</thead>
</table>
| Unit             | mORMot.pas | mORMot.pas       | mORMot.pas  | mORMotHttpServer.pasmORMot
                               |             |                  |                          | HttpClientModule.pas                  |
| Speed            | ****       | ***              | **          | *                     |
| Scaling          | ****       | *                | *           | ***                   |
| Hosting          | In-process | Local            | Local       | Remote                |
| Protocol         | Method call| WM_COPYDATA      | \pipe\mORMot_ | Standard              |
| Data             | JSON       | JSON             | JSON        | JSON                  |
| Run as service   | Stand alone| No               | Yes         | Yes                   |

Note that you can have *several* protocols exposing the same TSQLRestServer instance. You may expose the same server over HTTP and over named pipes, at the same time, depending on your speed requirements.
11.3. TSQLRest classes

This architecture is implemented by a hierarchy of classes, implementing the RESTful pattern - see REST (page 311) - for either stand-alone, client or server side, all inheriting from a TSQLRest common ancestor, as two main branches:

![Diagram of TSQLRest classes hierarchy]

RESTful Client-Server classes

All ORM operations (aka CRUD process) are available from the abstract TSQLRest class definition, which is overridden to implement either a Server (via TSQLRestServer classes), or a Client (via TSQLRestClientURI classes) access to the data.

You should instantiate the classes corresponding to the needed transmission protocol, but should better rely on abstraction, i.e. implement your whole code logic relying on abstract TSQLRestClient / TSQLRestServer classes. It will then help changing from one protocol or configuration at runtime, depending on your customer’s expectations.

11.3.1. Server classes

The following classes are available to implement a Server instance:

![Diagram of TSQLRestServer classes hierarchy]

RESTful Server classes

In practice, in order to implement the business logic, you should better create a new class, inheriting from one of the above TSQLRestServer classes. Having your own inherited class does make sense, especially for implementing your own method-based services - see below (page 373), or override internal methods.

The TSQLRestServerDB class is the main kind of Server of the framework. It will host a SQLite3 engine, as its core Database layer (page 196).

If your purpose is not to have a full SQLite3 engine available, you may create your server from a TSQLRestServerFullMemory class instead of TSQLRestServerDB: this will implement a fast in-memory engine (using TSQLRestStorageInMemory instances), with basic CRUD features (for ORM), and persistence on disk as JSON or optimized binary files - this kind of server is enough to handle authentication, and host services in a stand-alone way.
If your services need to have access to a remote ORM server, it may use a TSQLRestServerRemoteDB class instead: this server will use an internal TSQLRestClient instance to handle all ORM operations - it can be used e.g. to host some services on a stand-alone server, with all ORM and data access retrieved from another server: it will allow to easily implement a proxy architecture (for instance, as a DMZ for publishing services, but letting ORM process stay out of scope). See below (page 536) for some hosting scenarios.

Another option may be to use TSQLRestClientRedirect - see below (page 324) - which does something similar, but inheriting from TSQLRestClientURI.

11.3.2. Storage classes

In the mORMot units, you may also find those classes also inheriting from TSQLRestStorage:

```
TSQLRestStorageExternal
TSQLRestStorageInMemory
TSQLRestStorageInMemoryExternal
TSQLRestStorageMongoDB
TSQLRestStorageRecordBased
TSQLRestStorageRemote
```

**RESTful storage classes**

In the above class hierarchy, the TSQLRestStorage[InMemory][External] classes are in fact used to store some TSQLRecord tables in any non-SQL backend:
- TSQLRestStorageExternal maps tables stored in an external database - see External SQL database access (page 239);
- TSQLRestStorageInMemory stores the data in a TObjectList - see In-Memory "static" process (page 230);
- TSQLRestStorageRemote will redirect the CRUD operations of a given table to an external TSQLRest instance (client or server) - see Redirect to an external TSQLRest (page 233);
- TSQLRestStorageMongoDB will connect to a remote MongoDB server to store the tables as a NoSQL collection of documents - see External NoSQL database access (page 278).

Those classes are used within a main TSQLRestServer to host some given TSQLRecord classes, either in-memory, or on external databases. They do not enter in account in our Client-Server presentation, but are implementation details, on the server side.

11.3.3. Client classes

A full set of client classes will implement a RESTful access to a remote database, with associated services and business logic:
RESTful Client classes

Of course, all those TSQLRestClient* classes expect a TSQLRestServer to be available, via the corresponding transmission protocol.
11.4. In-process/stand-alone application

For a stand-alone application, create a TSQLRestClientDB. This particular class will initialize an internal TSQLRestServerDB instance, and you'll have full access to the SQLite3 database in the same process, with no speed penalty.

Content will still be converted to and from JSON, but there will be no delay due to the transmission of the data. Having JSON at hand will enable internal cache - see below (page 359) - and allow to combine this in-process direct process with other transmission protocols (like named pipes or HTTP).

Another option may be to use TSQLRestClientRedirect, which allows redirection from any TSQLRest class, either inheriting from TSQLRestClient or TSQLRestServer. Any TSQLRestClientURI.URI request will be passed to the redirected TSQLRest instance, which may be local or remote. The TSQLRestClientRedirect.RedirectTo method allows to enable or disable the redirection at runtime (by setting aRedirected=nil), or change the redirected TSQLRest instance on the fly, without creating a new TSQLRestClientRedirect instance.

You may also directly work with a TSQLRestServerDB instance, but you will miss some handy features of the TSQLRestClientURI class, like User-Interface interaction, or advanced ORM/SOA abilities, based on TSQLRestServer.URI process.

11.5. Local access via named pipes or Windows messages

For a Client-Server local application, that is some executable running on the same physical machine, create a TSQLRestServerDB instance, then use the corresponding ExportServer, ExportServerNamedPipe, ExportServerMessage method to instantiate either a in-process, Named-Pipe or Windows Messages server.

The Windows Messages layer has the lowest overhead and is the fastest transport layer available between several applications on the same computer. But it has the problem of being reserved to desktop applications (since Windows Vista), so you a Windows Messages server won't be accessible when run as a background service.

A named pipe communication is able to be served from a Windows service, and is known to be more efficient when transmitting big messages. So it is the preferred mean of communication for a local application sharing data between clients.

Due to security restriction of newer versions of Windows (i.e. starting with Vista), named pipes are not available by default over a network. This is the reason why this protocol is listed as local access mean only.
11.6. Network and Internet access via HTTP

For publishing a server via HTTP/1.1 over TCP/IP, creates a TSQLHttpServer instance, and associate your running TSQLRestServerDB to it.

Typical initialization code, as extracted from sample "04 - HTTP Client-Server", may be:

```pascal
Model := CreateSampleModel;
DBServer := TSQLRestServerDB.Create(Model, ChangeFileExt(paramstr(0), '.db3'), true);
DBServer.CreateMissingTables;
HttpServer := TSQLHttpServer.Create('8080', [DBServer], '+', HTTP_DEFAULT_MODE);
```

The following options is usually defined:

```pascal
HttpServer.AccessControlAllowOrigin := '*'; // allow cross-site AJAX queries
```

And you can optionally define some per domain / per sub-domain hosting redirection:

```pascal
HttpServer.DomainHostRedirect('project.com', 'root'); // 'root' is current Model.Root
HttpServer.DomainHostRedirect('blog.project.com', 'root/blog'); // MVC application
```

In all cases, even if HTTP protocol is very network friendly (especially over the 80 port), you shall always acquire IT approval and advices before any deployment over a corporate network, at least to negotiate firewall settings.

11.6.1. HTTP server(s)

The TSQLHttpServer class is able to use any of two HTTP server classes, as defined in SynCrtSock unit - and SynBidirSock for WebSockets:
- THttpServer which is a light and tuned server featuring a thread pool and IOCP implementation pattern, on the raw Sockets API;
- THttpApiServer which is based on http.sys API;
  - TWebSocketServer which is a THttpServer server, able to upgrade to the WebSockets protocol for asynchronous and bidirectional callbacks - see below (page 444).

```
TWebSocketServer
  └── THttpApiServer
    └── THttpServer
      └── THttpServerGeneric
          └── TThread
```

**THttpServerGeneric classes hierarchy**

On production, THttpApiServer seems to give the best results, and has a proven and secure implementation. It is also the only one class implementing HTTPS / SSL secure communication, if needed. That's why TSQLHttpServer will first try to use fastest http.sys kernel-mode server, then fail-back to the generic sockets-based THttpServer class in case of failure.

You can specify which kind of HTTP server class is to be used, via the aHttpServerKind: TSQLHttpServerOptions of the TSQLHttpServer.Create constructor. By default, it will be
HTTP_DEFAULT_MODE (i.e. useHttpApi over Windows), but you may specify useHttpApiRegisteringURI for automatic registration of the URI - see below (page 327) - or useHttpSocket to use the socket-based THttpServer, or useBidirSocket for TWebSocketServer.

The THttpServerGeneric abstract class provides one OnRequest property event, in which all high level process is to take place - it expects some input parameters, then will compute the output content to be sent as response:

```
TOnHttpServerRequest = function(Ctxt: THttpServerRequest): cardinal of object;
```

This event handler prototype is shared by both TThread classes instances able to implement a HTTP/1.1 server.

Both THttpApiServer and THttpServer classes will receive any incoming request, pass it to the TSQLRestServer instance matching the incoming URI request, via the OnRequest event handler.

If the request is a remote ORM operation, a JSON response will be retrieved from the internal cache of the framework, or computed using the SQLite3 database engine. In case of a remote service access - see below (page 372) - the request will be computed on the server side, also marshalling the data as JSON. If you specified useBidirSocket kind of server, you may use remote service access via interfaces, with asynchronous callbacks - see below (page 444).

The resulting JSON content will be compressed using our very optimized SynLZ algorithm (20 times faster than Zip/Deflate for compression), if the client is a Delphi application knowing about SynLZ - for an AJAX client, it won't be compressed by default (even if you can enable the deflate algorithm - which may slow down the server).

Then the response will be marked as to be sent back to the Client...

### 11.6.2. High-performance http.sys server

Since Windows XP SP2 and Windows Server 2003, the Operating System provides a kernel stack to handle HTTP requests. This http.sys driver is in fact a full featured HTTP server, running in kernel mode. It is part of the networking subsystem of the Windows operating system, as a core component.

The SynCrtSock unit can implement a HTTP server based on this component. Of course, the Synopse mORMot framework will use it. If it’s not available, it will launch our pure Delphi optimized HTTP server, using I/O completion ports and a Thread Pool.

What’s good about https.sys?

- **Kernel-mode request queuing**: Requests cause less overhead in context switching, because the kernel forwards requests directly to the correct worker process. If no worker process is available to accept a request, the kernel-mode request queue holds the request until a worker process picks it up.

- **Enhanced stability**: When a worker process fails, service is not interrupted; the failure is undetectable by the user because the kernel queues the requests while the WWW service starts a new worker process for that application pool.

- **Faster process**: Requests are processed faster because they are routed directly from the kernel to the appropriate user-mode worker process instead of being routed between two user-mode processes, i.e. the good old WinSock library and the worker process;

- **Embedded SSL process**, when secure HTTPS communication is needed.

#### 11.6.2.1. Use the http.sys server

Take a look at sample "04 - HTTP Client-Server", which is able to serve a SQLite3 database content...
over HTTP, using our RESTful ORM server. By default, it will try to use the http.sys server, then fall-back to plain socket server, in case of failure.

In fact, two steps are performed by the TSQLHttpServer constructor:
- The HTTP Server API is first initialized (if needed) during THttpApiServer.Create constructor call. The HttpApi.d11 library (which is the wrapper around http.sys) is loaded dynamically: so if you are running an old system (Windows XP SP1 for instance), you could still be able to use the server.
- It then tries to register the URI matching the RESTful model - REST (page 311) - via the THttpApiServer.AddUrl1 method. In short, the TSQLModel. Root property is used to compute the RESTful URI needed, just by the book. You can register several TSQLRestServer instances, each with its own TSQLModel1. Root, if you need it.

As we already stated, if any of those two steps fails (e.g. if http.sys is not available, or if it was not possible to register the URLs), the TSQLHttpServer class will fall back into using the other THttpServer class, which is a plain Delphi multi-threaded server. It won’t be said that we will let you down!

Inside http.sys all the magic is made… it will listen to any incoming connection request, then handle the headers, then check against any matching URL.

http.sys will handle all the communication by itself, leaving the server threads free to process the next request.

You can even use a special feature of http.sys to serve a file content as fast as possible. In fact, if you specify HTTP_RESP_STATICFILE as Ctxt.OutContentType, then Ctxt.OutContent is the UTF-8 file name of a file which must be sent to the client. Note that it will work only with THttpApiServer kind of server (i.e. using high performance http.sys API). But whole file access and sending will occur in background, at the kernel level, so with best performance. See sample "09 - HttpApi web server" and HttpApiServer.dpr file.

If you use a TSQLHttpServer, the easiest is to define a method-based service - see below (page 373) - and call Ctxt.ReturnFile() to return a file content from its name. We will see details about this below. Another possibility may be to override TSQLHttpServer.Request() method, as stated by Project04ServerStatic.dpr sample: but we think that a method-based service and Ctxt.ReturnFile() is preferred.

11.6.2.2. URI authorization as Administrator

This works fine under XP. Performances are very good, and stability is there. But… here comes the UAC nightmare again.

Security settings have changed since XP. Now only applications running with Administrator rights can register URLs to http.sys. That is, no real application. So the URI registration step will always fail with the default settings, under Vista and Seven.

The only case when authorization will be possible is when the application launched as a Windows Service, with default services execution user. By default, Windows services are launched with a User which has the Administrator rights.

11.6.2.2.1. Secure specific authorization

Standard security policy, as requested by Windows for all its http.sys based systems (i.e. IIS and WCF services) is to explicitly register the URI.
Depending on the system it runs on (i.e. Windows XP or Vista and up), a diverse command line tool is to be used. Can be confusing.

To keep it simple, our SynCrtSock unit provides a dedicated method to authorize a particular URI prefix to be registered by any user.

Therefore, a program can be easily created and called once with administrator rights to make http.sys work with our framework. This could be done, for instance, as part of your Setup program.

Then when your server application will be launched (for instance, as an application in tray icon with normal user rights, or a background Windows service with tuned user rights), it will be able to register all needed URL.

Here is a sample program which can be launched to allow our TestSQL3.dpr to work as expected - it will allow any connection via the 888 port, using TSQLModel. Root set as 'root' - that is, an URI prefix of http://+:888/root/ as expected by the kernel server:

```pascal
program TestSQL3Register;
uses
  SynCrtSock,
  SysUtils;

// force elevation to Administrator under Vista/Seven
{$R VistaAdm.res}

begin
  THttpApiServer.AddUrlAuthorize('root','888',false,'+');
end.
```

Take also a look at the Project04ServerRegister.dpr sample, in the context of a whole client/server RESTful solution over HTTP.

Note that you still need to open the IP port for incoming TCP traffic, in the Windows firewall, if you want your server to be accessible to the outer world, as usual.

### 11.6.2.2 Automatic authorization

An easier possibility could be to run the server application at least once as system Administrator.

The TSQLHttpServer.Create() constructor has a aHttpServerKind: TSQLHttpServerOptions parameter. By default, it will be set to useHttpApi. If you specify useHttpApiRegisteringURI, the class will register the URI before launching the server process.

All mORMot samples are compiled with this flag, as such:

```pascal
aHTTPServer := TSQLHttpServer.Create(PORT_NAME,[aServer],'+',useHttpApiRegisteringURI);
```

Note this does not follow default security policy of Windows. But it will make your application development easier.

### 11.6.2.2.3 Manual URI authorization

If you configured several http.sys servers on a given computer, you may have URI registration conflicts after some time.

You can use the netsh tool to list all registered URL with:

```cmd
netsh http show urlacl
```

You can optionally specify the fully qualified URL, for instance:
Then you can delete any url registration with:

```
netsh http delete urlacl url=http://host:port/[URI]
```

The [URI] is the registered URL path or domain. For instance:

```
netsh delete urlacl url=http://+:80/MyUri
netsh delete urlacl url=http://www.contoso.com:80/MyUri
```

Note that all those commands should be run with administrator user rights.


### 11.6.2.3. HTTP API 2.0 Features

Some THttpApi1Server methods are available with the HTTP Server 2.0 API, provided since Windows Vista and Windows Server 2008:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasAPI2</td>
<td>check if the HTTP API 2.0 is available</td>
</tr>
<tr>
<td>SetTimeOutLimits()</td>
<td>advanced timeout settings</td>
</tr>
<tr>
<td>LogStart() and LogStop</td>
<td>HTTP level standard logging</td>
</tr>
<tr>
<td>SetAuthenticationSchemes()</td>
<td>kernel-mode authentication</td>
</tr>
</tbody>
</table>

Please see the corresponding documentation of SynCrtSock.pas for further details, and https://msdn.microsoft.com/en-us/library/windows/desktop/aa364703 as low-level reference of these features. Note that our implementation of http.sys is more complete than the one currently included in the official .Net WCF framework. Not bad for a third-party library, isn't it?

### 11.6.3. HTTP client(s)

In fact, there are several implementation of a HTTP/1.1 clients, according to this class hierarchy:

```
TSQLHttpClientWinNet       TSQLHttpClientWinHTTP       TSQLHttpClientWebsockets
TSQLHttpClientWinGeneric   TSQLHttpClientWinSock        TSQLHttpClientCurl
TSQLHttpClientWinHTTP      TSQLHttpClientWinGeneric
TSQLHttpClientWinSock      TSQLHttpClientGeneric
```

So you can select either TSQLHttpClientWinSock, TSQLHttpClientWinINet or TSQLHttpClientWinHTTP for a HTTP/1.1 client, under Windows. By design, TSQLHttpClientWinINet or TSQLHttpClientWinHTTP are not available outside of Windows, but TSQLHttpClientCurl is a great option under Linux, if the libcurl library is installed, especially if you want to use HTTPS - it will call SynCurl.pas.
The TSQLHttpClientWebsockets class has the ability to upgrade the HTTP connection to the WebSockets protocol, which will be used for dual ways callbacks - see below (page 444).

Each class has its own architecture, and attaches itself to a Windows communication library, all based on WinSock API. As stated by their name, TSQLHttpClientWinSock will call directly the WinSock API, TSQLHttpClientWinINet will call WinINet API (as used by IE 6) and TSQLHttpClientWinHTTP will call the latest WinHTTP API:

- **WinSock** is the common user-space API to access the sockets stack of Windows, i.e. IP connection - it's able to handle any IP protocol, including TCP/IP, UDP/IP, and any protocol over it (including HTTP);
- **WinINet** was designed as an HTTP API client platform that allowed the use of interactive message dialogs such as entering user credentials - it's able to handle HTTP and FTP protocols;
  - WinHTTP's API set is geared towards a non-interactive environment allowing for use in service-based applications where no user interaction is required or needed, and is also much faster than WinINet - it only handles HTTP protocol.

**HTTP/1.1 Client architecture**

Here are some PROs and CONs of the available solutions, under Windows:

<table>
<thead>
<tr>
<th>Criteria</th>
<th>WinSock</th>
<th>WinINet</th>
<th>WinHTTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Level</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Local speed</td>
<td>Fastest</td>
<td>Slow</td>
<td>Fast</td>
</tr>
<tr>
<td>Network speed</td>
<td>Slow</td>
<td>Medium</td>
<td>Fast</td>
</tr>
<tr>
<td>Minimum OS</td>
<td>Win95/98</td>
<td>Win95/98</td>
<td>Win2000</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Not available</td>
<td>Available</td>
<td>Available</td>
</tr>
<tr>
<td>Integration with IE</td>
<td>None</td>
<td>Excellent (proxy)</td>
<td>Available (see below)</td>
</tr>
<tr>
<td>User interactivity</td>
<td>None</td>
<td>Excellent (authentication, dial-up)</td>
<td>None</td>
</tr>
</tbody>
</table>

As stated above, there is still a potential performance issue to use the direct TSQLHttpClientWinSock class over a network. It has been reported on our forum, and root cause was not identified yet.

Therefore, the TSQLHttpClient class maps by default to the TSQLHttpClientWinHTTP class. This is the recommended usage from a Delphi client application.

Note that even if WinHTTP does not share by default any proxy settings with Internet Explorer, it can import the current IE settings. The WinHTTP proxy configuration is set by either proxycfg.exe on Windows XP and Windows Server 2003 or earlier, or netsh.exe on Windows Vista and Windows
Server 2008 or later; for instance, you can run "proxycfg -u" or "netsh winhttp import proxy source=ie" to use the current user's proxy settings for Internet Explorer. Under 64-bit Vista/Seven, to configure applications using the 32 bit WinHttp settings, call netsh or proxycfg bits from %SystemRoot%\SysWOW64 folder explicitly.

Note that by design, the TSQLHttpClient* classes, like other TSQLRestClientURI implementations, were designed to be thread safe, since their URI() method is protected by a lock. See below (page 335).

11.6.4. HTTPS server

The http.sys kernel mode server can be defined to serve HTTPS secure content, i.e. the SSL protocol over HTTP.

When the aHttpServerSecurity parameter is set to secSSL for the TSQLHttpServer.Create() constructor, the SSL layer will be enabled within http.sys. Note that useHttpSocket kind of server does not offer SSL/TLS encryption yet.

In order to let the SSL layer work as expected, you need first to create and import a set of certificates.

11.6.4.1. Certificates

You need one certificate (cert) to act as your root authority, and one to act as the actual certificate to be used for the SSL, which needs to be signed by your root authority. If you don't set up the root authority your single certificate won't be trusted, and you will start to discover this through a series of extremely annoying exceptions, long after the fact. To get a free certificate, i.e. for testing purposes, you may use an online service like http://www.startssl.com..

Depending on the Windows revision you are using, you can run the Internet Information Services (IIS) Manager: from the Windows Start menu, click Administrative Tools > Internet Information Services (IIS) Manager. See http://support.microsoft.com/kb/299875..

You could also install the needed certificate by using some command lines - this may be handy for fast installation using a .bat file. Here are the needed steps, as detailed in http://www.codeproject.com/Articles/24027/SSL-with-Self-hosted-WCF-Service.. and http://msdn.microsoft.com/en-us/library/ms733791..

The following command (run in a Visual Studio command prompt) will create your root certificate:

```
makecert -sv SignRoot.pvk -cy authority -r signroot.cer -a
    sha1 -n "CN=Dev Certification Authority" -ss my -sr localmachine
```

Take a look at the above links to see what each of these arguments mean, it isn't terribly important, but it's nice to know.

The MakeCert tool is available as part of the Windows SDK, which you can download from http://go.microsoft.com/fwlink/p/?linkid=84091.. if you do not want to download the whole Visual Studio package. Membership in Administrators, or equivalent, on the local computer is the minimum required to complete this procedure.

Once this command has been run and succeeded, you need to make this certificate a trusted authority. You do this by using the MMC snap in console. Go to the run window and type "mmc", hit enter. Then in the window that opens (called the "Microsoft Management Console", for those who care) perform the following actions:

File -> Add/Remove Snap-in -> Add... -> Double click Certificates -> Select Computer Account and Click
Then select the Certificates (Local Computer) -> Personal -> Certificates node.

You should see a certificate called "Dev Certificate Authority" (or whatever else you decided to call it as parameter in the above command line). Move this certificate from the current node to Certificates (Local Computer) -> Trusted Root Certification Authorities -> Certificates node, drag and drop works happily.

Now you have NOT the cert you need :) You have made yourself able to create trusted certs though, which is nice. Now you have to create another cert, which you are actually going to use. 

Run makecert again, but run it as follows...

```
makecert -iv SignRoot.pvk -ic signroot.cer -cy end -pe -n
  CN="localhost" -eku 1.3.6.1.5.5.7.3.1 -ss my -sr
  localmachine -sky exchange -sp
"Microsoft RSA SChannel Cryptographic Provider" -sy 12
```

Note that you are using the first certificate as the author for this latest one. This is important... where I have localhost you need to put the DNS name of your box. In other words, if you deploy your service such that its endpoint reads http://bob:10010/Service then the name needs to be bob. In addition, you are going to need to do this for each host you need to run as (yes, so one for bob and another one for localhost).

Get the signature of your cert by double clicking on the cert (Select the Certificates (Local Computer) ' Personal ' Certificates), opening the details tab, and scrolling down to the "Thumbprint" option.

Select the thumbprint and copy it. Put it in Notepad or any other text editor and replace the spaces with nothing. Keep this thumbprint heaxdecimal value safe, since we will need it soon.

You have your certs set up. Congrats! But we are not finished yet.

**11.6.4.2. Configure a Port with an SSL certificate**

Now you get to use another fun tool, httpcfg (for XP/2003), or its newer version, named aka netsh http (for Vista/Seven/Eight).

Firstly run the command below to check that you don't have anything running on a port you want.

```
httpcfg query ssl
```

(under XP)

```
netsh http show sslcert
```

(under Vista/Seven/Eight)

If this is your first time doing this, it should just return a newline. If there is already SSL set up on the exact IP you want to use (or if later on you need to delete any mistakes) you can use the following command, where the IP and the port are displayed as a result from the previous query.

Now we have to bind an SSL certificate to a port number, as such (here below, 0000000000003ed9cd0c315bbb6dc1c08da5e6 is the thumbprint of the certificate, as you copied it into the notepad in the previous paragraph):

```
httpcfg set ssl -i 0.0.0.0:8012 -h 0000000000003ed9cd0c315bbb6dc1c08da5e6
```
11.6.5 Custom Encodings

11.6.5.1 SynLZ/deflate compression

On the client side, the TSQLHttpClientGeneric.Compression property is by default set as such:

```
MyClient.Compression := [hcSynLZ];
```

It will enable SynLZ compression in the HTTP headers:

```
ACCEPT-ENCODING: synlz
```

Our SynLZ is efficient, especially on JSON content, and very fast on the server side. It will therefore use less resources than hcDeflate, so may be preferred when balancing the resource / concurrent client ratio.

You may include hcDeflate to the property, if you want to support this zip-derivated compression algorithm, e.g. from browsers or any HTTP library. In terms of CPU resources, hcDeflate will be more consuming than hcSynLZ, but will obtain a slightly better compression ratio.

If both [hcSynLZ, hcDeflate] are defined, mORMot clients will use SynLZ compression, while other clients (e.g. browsers which do not know about the SynLZ encoding), will use the standard deflate compression.

11.6.5.2 AES encryption over HTTP

In addition to regular HTTPS flow encryption, which is not easy to setup due to the needed certificates, mORMot proposes a proprietary encryption scheme. It is based on SHA256 and AES256-CFB algorithms, so is known to be secure. You do not need to setup anything on the server or the client configuration, just run the TSQLHttpClient and TSQLHttpServer classes with the corresponding parameters.

Note that this encryption uses a global key for the whole process, which should match on both Server and Client sides. You should better hard-code this public key in your Client and Server Delphi applications, with some variants depending on each end-user service. You can use CompressShaAesSetKey() as defined in SynCrypto.pas to set globally this Encryption Key, and an optional Initialization Vector. You can even customize the AES chaining mode, if the default TAESCFB
mode is not what you expect.

When the aHttpServerSecurity parameter is set to secSynShaAes for the TSQLHttpServer.Create() constructor, this proprietary encryption will be enabled on the server side. For instance:

```delphi
MyServer := TSQLHttpServer.Create('888',[DataBase],'+',useHttpApi,32,secSynShaAes);
```

On the client side, you can just set the TSQLHttpClientGeneric.Compression property as expected:

```delphi
MyClient.Compression := [hcSynShaAes];
```

Once those parameters have been set, a new proprietary encoding will be defined in the HTTP headers:

```http
ACCEPT-ENCODING: synshaaes
```

Then all HTTP body content will be compressed via our SynLZ algorithm, and encoded using the very secure AES256-CFB scheme. On both client and server side, this encryption will use AES-NI hardware instructions, if available on the CPU it runs on. It ensures that security is enhanced not at the price of performance and scalability.

Since it is a proprietary algorithm, it will work only for Delphi clients. When accessing for a plain AJAX client, or a Delphi application with TSQLHttpClientGeneric.Compression = [], there won't be any encryption at all, due to way HTTP accepts its encoding. For safety, you should therefore use it in conjunction with per-URI Authentication - see below (page 546).

### 11.6.5.3. Prefer WebSockets between mORMot nodes

As we just saw, defining hcSynShaAes will only be available between mORMot nodes, if both do support the encoding. There is no insurance that content will be encrypted during transmission, e.g. if the client did not define synshaaes.

Therefore, for truly safe communication between mORMot nodes, you may consider our WebSockets client/server implementation instead - see below (page 444). It implements a proprietary binary protocol for its communication frames, using also SynLZ compression and AES256-CFB encryption. And, last but not least, it features real-time callbacks, if needed. This kind of access may in fact be considered as the safest available mean of remote connection to a mORMot server, from stable mORMot clients, e.g. in a mORMot Cloud. Then RESTful (AJAX/mobile) clients, may rely on plain HTTP, with hcDeflare compression.
11.7. Thread-safety

We tried to make mORMot at the same time fast and safe, and able to scale with the best possible performance on the hardware it runs on. Multi-threading is the key to better usage of modern multi-core CPUs, and also client responsiveness.

As a result, on the Server side, our framework was designed to be thread-safe.

On typical production use, the mORMot HTTP server - see JSON RESTful Client-Server (page 295) - will run on its own optimized thread pool, then call the TSQLRestServer.URI method. This method is therefore expected to be thread-safe, e.g. from the TSQLHttpServer.Request method. Thanks to the RESTful approach of our framework, this method is the only one which is expected to be thread-safe, since it is the single entry point of the whole server. This KISS design ensure better test coverage.

On the Client side, all TSQLRestClientURI classes are protected by a global mutex (Critical Sections), so are thread-safe. As a result, a single TSQLHttpClient instance can be shared among several threads, even if you may also use one client per thread, as is done with sample 21 - see below, for better responsiveness.

11.7.1. Thread safe design

We will now focus on the server side, which is the main strategic point (and potential bottleneck or point of failure) of any Client-Server architecture.

In order to achieve this thread-safety without sacrificing performance, the following rules were applied in TSQLRestServer.URI:

- Most of this method's logic is to process the URI and parameters of the incoming request (in TSQLRestServerURIContext.URI Decode* methods), so is thread-safe by design (e.g. Model and RecordProps access do not change during process);
- At RESTful / CRUD level, Add/Update/Delete/TransactionBegin/Commit/Rollback methods are locked by default (with a 2 seconds timeout), and Retrieve* methods are not;
- TSQLRestStorage main methods (EngineList, EngineRetrieve, EngineAdd, EngineUpdate, EngineDelete, EngineRetrieveBlob, EngineUpdateBlob) are thread-safe: e.g. TSQLRestStorageInMemory uses a per-Table Critical Section;
- TSQLRestServerCallback method-based services - i.e. published methods of the inherited TSQLRestServer class as stated below (page 373) - must be implemented to be thread-safe by default;
- Interface-based services - see below (page 419) - have several execution modes, including thread safe automated options (see TServiceMethodOption) or manual thread safety expectation, for better scaling - see below (page 461);
- A protected fSessionCriticalSection is used to protect shared fSession[] access between clients;
- The SQLite3 engine access is protected at SQL/JSON cache level, via DB.LockJSON() calls in TSQLRestServerDB methods;
- Remote external tables - see External SQL database access (page 239) - use thread-safe connections and statements when accessing the databases via SQL;
- Access to fStats was not made thread-safe, since this data is indicative only: a mutex was not used to protect this resource.

We tried to make the internal Critical Sections as short as possible, or relative to a table only (e.g. for...
At SQLite3 engine level, there is some kind of "giant lock", so all TSQLDatabase requests process will be queued. This induces only a slight performance penalty - see Data access benchmark (page 199) - since the internal SQL/JSON cache implementation needs such a global lock, and since most of the SQLite3 resource use will consist in disk access, which gains to be queued. It also allows to use the SQLite3 engine in lmExclusive locking mode if needed - see ACID and speed (page 222) - with both benefits of high performance and multi-thread friendliness.

From the Client-side, the REST core of the framework is expected to be Client-safe by design, therefore perfectly thread-safe: it is one benefit of the stateless architecture.

### 11.7.2. Advanced threading settings

You can use TSQLRestServerURI.AcquireExecutionMode[] property to refine the server-side threading mode. When amLocked is set, you can also set the AcquireExecutionLockedTimeOut[] property to specify a wait time to acquire the lock.

The default threading behavior is the following:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>execSOAByMethod</td>
<td>for method-based services</td>
<td>amUnlocked</td>
</tr>
<tr>
<td>execSOAByInterface</td>
<td>for interface-based services</td>
<td>amUnlocked</td>
</tr>
<tr>
<td>execORMGet</td>
<td>for ORM reads i.e. Retrieve* methods</td>
<td>amUnlocked</td>
</tr>
<tr>
<td>execORMWrite</td>
<td>for ORM writes i.e. Add Update Delete TransactionBegin Commit Rollback methods</td>
<td>amLocked + timeout of 2000 ms</td>
</tr>
</tbody>
</table>

On need, you can change those settings, to define a particular execution scheme. For instance, some external databases (like MS SQL) expect any transaction to be executed within the same connection, so in the same thread context for SynOleDB.pas, since it uses a per-thread connection pool. When the server is remotely access via HTTP, the incoming requests will be executed from any thread of the HTTP server thread pool. As a result, you won’t be able to manage a transaction over MS SQL from the client-side with the default settings.

To fix it, you can ensure all ORM write operations will be executed in a dedicated background thread, by setting either:

```pascal
aServer.AcquireExecutionMode[execORMWrite] := amBackgroundThread;
aServer.AcquireWriteMode := amBackgroundThread; // same as previous
```

The same level of thread-safety can be defined for all kind of commands, even if you should better know what you are doing when changing the default settings, since it may create some giant locks on the server side, therefore voiding any attempt to performance scaling via multi-threading - which is what mORMot excels in.

At ORM level, with external databases, your mORMot server may suffer from broken connection to the remote database. To avoid this, you may use ConnectionTimeOutMinutes property to specify a maximum period of inactivity after which all connections will be flushed and recreated, to avoid potential broken connections issues.

In this case, you should ensure that all ORM process is blocked so that clearing the connection pool won’t break anything in your multi-threaded server. As such, you may set a blocking mode for both execORMGet and execORMWrite, for instance:
The above commands will create one thread for all read operations (execORMGet), and another thread for all write operations (execORMWrite). If you want all database access to take place in a single thread, for both read and write operations, you could write:

```pascal
aServer.AcquireExecutionMode[execORMGet] := amBackgroundORMSharedThread;
aServer.AcquireExecutionMode[execORMWrite] := amBackgroundORMSharedThread;
```

For instance, this sounds mandatory when using Jet/MSAccess as external database, since its implementation seems not thread-safe: if you write in one thread, then read immediately in another thread, the Jet engine is not able to find the just written data from the 2nd thread. This is clearly a bug of the Jet engine - but setting amBackgroundORMSharedThread option to circumvent the issue.

During any ORM or SOA process, you can access the current execution context from the ServiceContext threadvar variable, as stated below (page 432). For instance, you can retrieve the current logged user, or its session ID.

In practice, execSOAByMethod may benefit of a per-method locking, execSOAByInterface of using its own execution options - see below (page 461), and execORMGet to be let unlocked to allow concurrent reads of all connected clients.

### 11.7.3. Proven behavior

When we are talking about thread-safety, nothing compares to a dedicated stress test program. An average human brain (like ours) is not good enough to ensure proper design of such a complex process. So we have to prove the abilities of our little mORMot.

In the supplied regression tests, we designed a whole class of multi-thread testing, named TTestMultiThreadProcess. Its methods will run every and each Client-Server protocols available (direct access via TSQLRestServerDB or TSQLRestClientDB, Windows Messages, named pipes, and both HTTP servers - i.e. http.sys based or WinSock-based)- see Client-Server process (page 318).

Each protocol will execute in parallel a list of INSERTs - i.e. TSQLRest.Add() - followed by a list of SELECTs - i.e. TSQLRest.Retrieve(). Those requests will be performed in 1 thread, then 2, 5, 10, 30 and 50 concurrent threads. The very same SQLite3 database (in lmExclusive locking mode) is accessed at once by all those clients. Then the IDs generated by each thread are compared together, to ensure no cross-insertion did occur during the process.

Those automated tests did already reveal some issues in the initial implementation of the framework. We fixed any encountered problems, as soon as possible. Feel free to send us any feedback, with code to reproduce the issue: but do not forget that multi-threading is also difficult to test - problems may occur not in the framework, but in the testing code itself!

When setting OperationCount to 1000 instead of the default 200, i.e. running 1000 INSERTions and 1000 SELECTs in concurrent threads, the numbers are the following, on the local machine (compiled with Delphi XE4):

<table>
<thead>
<tr>
<th>Multi thread process:</th>
<th>1 assertion passed 3.11ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Create thread pool:</td>
<td>1 assertion passed 3.11ms</td>
</tr>
<tr>
<td>- TSQLRestServerDB:</td>
<td>24,061 assertions passed 903.31ms</td>
</tr>
<tr>
<td>1=41986/s 2=24466/s 5=14041/s 10=9212/s 30=18376/s 50=10028/s</td>
<td></td>
</tr>
<tr>
<td>- TSQLRestClientDB:</td>
<td>24,062 assertions passed 374.93ms</td>
</tr>
<tr>
<td>1=38606/s 2=35823/s 5=30083/s 10=32739/s 30=33454/s 50=30905/s</td>
<td></td>
</tr>
<tr>
<td>- TSQLRestClientURINamedPipe:</td>
<td>12,012 assertions passed 1.68s</td>
</tr>
<tr>
<td>1=4562/s 2=5062/s 5=3177/s</td>
<td></td>
</tr>
<tr>
<td>- TSQLRestClientURIMessage:</td>
<td>16,022 assertions passed 616.00ms</td>
</tr>
</tbody>
</table>
For direct in-process access, TSQLRestClientDB sounds the best candidate: its abstraction layer is very thin, and much more multi-thread friendly than straight TSQLRestServerDB calls. It also will feature a cache, on need - see ORM Cache (page 174). And it will allow your code to switch between TSQLRestClientURI kind of classes, from its shared abstract methods.

Named pipes and Windows Messages are a bit constrained in highly parallel mode, but HTTP does pretty good. The server based on http.sys (HTTP API) is even impressive: the more clients, the more responsive it is. It is known to scale much better than the WinSock-based class supplied, which shines with one unique local client (i.e. in the context of those in-process regression tests), but sounds less reliable on production.

11.7.4. Highly concurrent clients performance

In addition, you can make yourself an idea, and run the "21 - HTTP Client-Server performance" sample programs, locally or over a network, to check the mORMot abilities to scale and serve a lot of clients with as few resources as possible.

Compile both client and server projects, then launch Project21HttpServer.exe. The server side will execute as a console window.

This Server will define the same TSQLRecordPeople as used during our multi-thread regression tests, that is:

```pascal
type
  TSQLRecordPeople = class(TSQLRecord)
  private
    fFirstName: RawUTF8;
    fLastName: RawUTF8;
    fYearOfBirth: integer;
    fYearOfDeath: word;
  published
    property FirstName: RawUTF8 read fFirstName write fFirstName;
    property LastName: RawUTF8 read fLastName write fLastName;
    property YearOfBirth: integer read fYearOfBirth write fYearOfBirth;
    property YearOfDeath: word read fYearOfDeath write fYearOfDeath;
  end;

T
```

The server main block is just the following:

```pascal
aModel := TSQLModel.Create([TSQLRecordPeople]);
try
  aDatabaseFile := ChangeFileExt(paramstr(0),'.db3');
  DeleteFile(aDatabaseFile);
  aServer := TSQLRestServerDB.Create(aModel,aDatabaseFile);
  try
    aServer.DB.Synchronous := smOff;
    aServer.DB.LockingMode := lmExclusive;
    aServer.NoAJAXJSON := true;
    aServer.CreateMissingTables;
    // launch the server
    aHTTPServer := TSQLHttpServer.Create('888',[aServer]);
  try
    writeln('#13#10'Background server is running at http://localhost:888'#13#10+
      '#13#10'Press [Enter] to close the server.');
    ConsoleWaitForEnterKey;
  finally
```
Following the Model-View-Controller (page 86) pattern, aServer will give remote CRUD access to the TSQLRecordPeople table (as defined in aModel), from HTTP. We defined Synchronous := smOff and LockingMode := lmExclusive to have the best performance possible, as stated by ACID and speed (page 222). Our purpose here is not to have true ACID behavior, but test concurrent remote access.

The Client is just a RAD form which will execute the very same code than during the regression tests, i.e. a TTestMultiThreadProcess class instance, as shown by the following code:

```pascal
Tests := TSynTestsLogged.Create;
Test := TTestMultiThreadProcess.Create(Tests);
try
    Test.ClientOnlyServerIP := StringToAnsi7(lbledtServerAddress.Text);
    Test.MinThreads := ThreadCount;
    Test.MaxThreads := ThreadCount;
    Test.OperationCount := OperationCount;
    Test.ClientPerThread := ClientPerThread;
    Test.CreateThreadPool;
    txt := Format('%s',#13#10'Test started with %d threads, %d client(s) per thread and %d rows to be inserted...');
    [txt,ThreadCount,ClientPerThread,OperationCount];
    mmoInfo.Text := txt;
    Timer.Start;
    Test._TSQLHttpClientWinHTTP_HTTPAPI;
    txt := mmoInfo.Text+Format(#13#10'Assertion(s) failed: %d / %d'
        #13#10'Number of clients connected at once: %d'
        #13#10'Time to process: %s'
        #13#10'Operation per second: %d',
        [Test.AssertionsFailed,TestAssertions,
        ThreadCount*ClientPerThread,Timer.Stop,Timer.PerSec(OperationCount*2)]);
    mmoInfo.Text := txt;
finally
    Test.Free;
    Tests.Free;
end;
```

Each thread of the thread pool will create its own HTTP connection, then loop to insert (Add ORM method) and retrieve (Retrieve ORM method) a fixed number of objects - checking that the retrieved object fields match the inserted values. Then all generated IDs of all threads are checked for consistency, to ensure no race condition did occur.

The input parameters are therefore the following:
- Remote HTTP server IP (port is 888);
- Number of client threads;
- Number of client instances per thread;
- Number of TSQLRecordPeople objects added.

When running over the following hardware configuration:
- Server is a Core i7 Notebook, with SSD, under Windows 7;
- Client is a Core 2 Duo Workstation, with regular hard-drive (not used), under Windows 7;
- Communicating over a somewhat slow 100 Mb network with a low priced Ethernet HUB.

Typical results are the following:
During all tests, no assertion failed, meaning that no concurrency problem did occur, nor any remote command lost. The SQLite3 core, exposes via the mORMot server, outputs data at an amazing pace of 6000 op/sec - i.e. comparable to most high-end databases. It is worth noting that when run several times in a row, the same set of input parameters give the very same speed results: it indicates that the architecture is pretty stable and could be considered as safe. The system is even able to serve 50000 connected clients at once, with no data loss - in this case, performance is lower (2152 insert/second in the above table), but we clearly reached the CPU and network limit of our client hardware configuration; in the meanwhile, server CPU resources on the Notebook server did have still some potential, and RAM consumption was pretty slow.

Average performance is pretty good, even more if we consider that we are inserting one object per request, with no transaction. In fact, it sounds like if our little SQLite3 server is faster than most database servers, even when accessed in highly concurrent mode! In batch mode - see below (page 350) - we may achieve amazing results.

Feel free to send your own benchmark results and feedback, e.g. with concurrent clients on several workstations, or long-running tests, on our forums.
12. Client-Server ORM

As stated above, all ORM features can be accessible either stand-alone, or remotely via some dedicated Client-Server process (page 318).

That is, CRUD operations can be executed either at the database level, or remotely, from the same methods defined in TSQLRest abstract class.

This feature has several benefits, among them:
- No need to deploy the database client library for your application clients - a standard IP network connection is enough;
- Therefore the client application can safely remain small, and stand-alone - no installation step is necessary, and you still have the full power of a native rich client;
- Clients access their objects in an abstract way, i.e. without any guess on how persistence is handled: some classes may be stored in one SQLite3 database, others may exist only in server’s memory, others may be stored e.g. in an external Oracle, Firebird, PostgreSQL, MySQL, DB2, Informix or MS SQL database;
- You can switch from local to remote access just by changing the class type, even at runtime;
- Optimization is implemented at every level of the n-Tier architecture, e.g. cache or security.

12.1. ORM as local or remote

Typical Client-Server RESTful POST / Add request over HTTP/1.1 will be implemented as such, on both Client and Server side:
Of course, several clients can access to the same server.
The same server is also able to publish its RESTful services over several communication protocol at once, e.g. HTTP/1.1 for remote access over a network (either corporate or the Internet), named pipes or Windows Messages for fast local access.

The above diagram describes a direct INSERT into the Server’s main SQLite3 engine, but other database back-ends are available - see Database layer (page 196).

It is possible to by-pass the whole Client-Server architecture, and let the application be stand-alone, by defining a TSQLRestClientDB class, which will embed a TSQLRestServerDB instance in the same executable:

---

**Client-Server implementation - Stand-Alone application**

In fact, the same executable could be launched as server, as stand-alone application, or even client application! It is just a matter of how you initialize your TSQLRest classes instances - see Client-Server process (page 318). Some mORMot users use this feature to ease deployment, support and configuration. It can be also extremely useful at debugging time, since you may run the server and client side of your project at once within the same application, from the IDE.
In case of a Virtual Table use (either in-memory or for accessing an external database), the client side remains identical. Only the server side is modified as was specified by *External database ORM internals* (page 273):

Client-Server implementation - Server side with Virtual Tables

In fact, the above function correspond to a database model with only external virtual tables, and with `StaticVirtualTableDirect=false`, i.e. calling the Virtual Table mechanism of *SQLite3* for each
request.

But most of the time, i.e. for RESTful / CRUD commands, the execution is more direct:

As stated in *External SQL database access* (page 239), the static `TSQLRestStorageExternal` instance is called for most RESTful access. In practice, this design will induce no speed penalty, when compared to a direct database access. It could be even faster, if the server is located on the same computer than the database: in this case, use of JSON and REST could be faster - even faster when using below (page 350).

In order to be exhaustive, here is a more complete diagram, showing how native SQLite3, in-memory or external tables are handled on the server side. You'll find out how CRUD statements are handled directly for better speed, whereas any SQL JOIN query can also be processed among all kind of tables.
You will find out some speed numbers resulting from this unique architecture in the supplied Data access benchmark (page 199).
12.2. Stateless design

12.2.1. Server side synchronization

Even if Stateless ORM (page 314), it’s always necessary to have some event triggered on the server side when a record is edited.

On the server side, you can use this method prototype:

```pascal
type
  /// used to define how to trigger Events on record update
  /// - see TSQLRestServer.OnUpdateEvent property
  /// returns true on success, false if an error occurred (but action must continue)
  TNotifySQLEvent = function(Sender: TSQLRestServer; Event: TSQLEvent; aTable: TSQLRecordClass; aID: TID): boolean of object;

TSQLRestServer = class(TSQLRest)
(...)
  /// a method can be specified here to trigger events after any table update
  OnUpdateEvent: TNotifySQLEvent;
```

12.2.2. Client side synchronization

But if you want all clients to be notified from any update, there is no direct way of broadcasting some event from the server to all clients.

It’s not even technically possible with pipe-oriented transport layer, like named pipes or the TCP/IP - HTTP protocol.

What you can do easily, and is what should be used in such case, is to have a timer in your client applications which will call TSQLRestClientURI. UpdateFromServer() method to refresh the content of any TSQLRecord or TSQLTableJSON instance:

```pascal
/// check if the data may have changed of the server for this objects, and
/// update it if possible
/// - only working types are TSQLTableJSON and TSQLRecord descendants
/// - make use of the InternalState function to check the data content revision
/// - return true if Data is updated successfully, or false on any error
/// during data retrieval from server (e.g. if the TSQLRecord has been deleted)
/// - if Data contains only one TSQLTableJSON, PCurrentRow can point to the
/// current selected row of this table, in order to refresh its value
function UpdateFromServer(const Data: array of TObject; out Refreshed: boolean;
  PCurrentRow: PInteger = nil): boolean;
```

With a per-second timer, it’s quick and reactive, even over a remote network.

The stateless aspect of REST allows this approach to be safe, by design.

This is handled natively by our Client User Interface classes, with the following parameter defining the User interface:

```pascal
/// defines the settings for a Tab
TSQLRibbonTabParameters = object
  (...)
  /// by default, the screens are not refreshed automatically
  /// but you can enable the auto-refresh feature by setting this
  /// property to TRUE, and creating a WM_TIMER timer to the form
  AutoRefresh: boolean;
```

This parameter will work only if you handle the WM_TIMER message in your main application form, and call Ribbon.WMRefreshTimer.
See for example this method in the main demo (FileMain.pas unit):

```pascal
procedure TForm1.WMRefreshTimer(var Msg: TWMTimer);
begin
Ribbon.WMRefreshTimer(Msg);
end;
```

In a multi-threaded client application, and even on the server side, a stateless approach makes writing software easier. You do not have to care about forcing data refresh in your client screens. It's up to the screens to get refreshed. In practice, I found it very convenient to rely on a timer instead of calling the somewhat "delicate" TThread. Synchronize method.

### 12.2.3. Let applications be responsive

All the client communication is executed by default in the current thread, i.e. the main thread for a typical GUI application.

Since all communication is performed in blocking mode, if the remote request takes long to process (due to a bad/slow network, or a long server-side action), the application may become unresponsive, from the end-user experience. Even Windows may be complaining about a "non responsive application", and may propose to kill the process, which is far away from an expected behavior.

In order to properly interacts with the user, a OnIdle property has been defined in TSQLRestClientURI, and will change the way communication is handled. If a callback event is defined, all client communication will be processed in a background thread, and the current thread (probably the main UI thread) will wait for the request to be performed in the background, running the OnIdle callback in loop in the while.

You can find in the mORMotUILogin unit two methods matching this callback signature:

```pascal
TLoginForm = class(TForm)
(....)
class procedure OnIdleProcess(Sender: TSynBackgroundThreadAbstract; ElapsedMS: Integer);
class procedure OnIdleProcessForm(Sender: TSynBackgroundThreadAbstract; ElapsedMS: Integer);
end;
```

The first OnIdleProcess() callback will change the mouse cursor shape to crHourClass after a defined period of time. The OnIdleProcessForm() callback won't only change the mouse cursor, but also display a pop-up window with a 'Please wait...' message, if the request takes even more time. Both will call Application.ProcessMessages to ensure the application User Interface is still responsive.

Some global variable were also defined to tune the behavior of those two callbacks:

```pascal
var
    /// define when TLoginForm.OnIdleProcess() has to display the crHourGlass cursor
    /// after a given time elapsed, in milliseconds
    /// - default is 100 ms
    OnIdleProcessCursorChangeTimeout: integer = 100;

    /// define when TLoginForm.OnIdleProcessForm() has to display the temporary
    /// form after a given time elapsed, in milliseconds
    /// - default is 2000 ms, i.e. 2 seconds
    OnIdleProcessTemporaryFormTimeout: integer = 2000;

    /// define the message text displayed by TLoginForm.OnIdleProcessForm()
    /// - default is sOnIdleProcessFormMessage resourcestring, i.e. 'Please wait...'
    OnIdleProcessTemporaryFormMessage: string;
```

You can therefore change those settings to customize the user experience. We tested it with a 3 second artificial temporizer for each request, and the applications were running smoothly, even if
slowly - but comparable to most Web Applications, in fact. The SynFile main demo (available in the SQLite3\Samples\MainDemo folder) defines such a callback.

Note that this OnIdle feature is defined at TSQLRestClientURI class level, so is available for all communication protocols, not only HTTP but named pipes or in-process, so could be used to enhance user experience in case of some time consuming process.
12.3. BATCH sequences for adding/updating/deleting records

12.3.1. BATCH process

When use the so-called BATCH sequences?

In a standard Client-Server architecture, especially with the common understanding (and most implementations) of a RESTful service, any Add / Update / Delete method call requires a back and forth flow to then from the remote server. A so-called round-trip occurs: a message is sent to the client, the a response is sent back to the client.

In case of a remote connection via the Internet (or a slow network), you could have up to 100 ms of latency: it's just the "ping" timing, i.e. the time spent for your IP packet to go to the server, then back to you.

If you are making a number of such calls (e.g. add 1000 records), you'll have 100*1000 ms = 100 s = 1:40 min just because of this network latency!

The BATCH sequence allows you to regroup those statements into just ONE remote call. Internally, it builds a JSON stream, then post this stream at once to the server. Then the server answers at once, after having performed all the modifications.

Some new TSQLRestClientURI methods have been added to implement BATCH sequences to speed up database modifications: after a call to BatchStart, database modification statements are added to the sequence via BatchAdd / BatchUpdate / BatchDelete, then all statements are sent as one to the remote server via BatchSend - this is MUCH faster than individual calls to Add / Update / Delete in case of a slow remote connection (typically HTTP over Internet).

Since the statements are performed at once, you can't receive the result (e.g. the ID of the added row) on the same time as you append the request to the BATCH sequence. So you'll have to wait for the BatchSend method to retrieve all results, at once, in a dynamic array of TID.
As you may guess, it's also a good idea to use a transaction for the whole process. By default, the BATCH sequence is not embedded into a transaction.

You have two possibilities to add a transaction:
- Either let the caller use an explicit TransactionBegin... try... Commit except RollBack block;
- Or specify a number of rows as AutomaticTransactionPerRow parameter to BatchStart(): in this case, a transaction will be emitted (up to the specified number of rows) on the server side. You can just set maxInt if you want all rows to be modified in a single transaction.

This second method is preferred, since defining transactions from the client side is not a good idea: it may block other clients attempts to create their own transaction.

Here is typical use (extracted from the regression tests in SynSelfTests.pas:

```
// start the BATCH sequence
Check(ClientDist.BatchStart(TSQLRecordPeople,1000));
// now a transaction will be created by chunk of 1000 modifications
// delete some elements
for i := 0 to n-1 do
  Check(ClientDist.BatchDelete(IntArray[i])=i);
// update some elements
nupd := 0;
for i := 0 to aStatic.Count-1 do
  if i and 7<>0 then
    begin // not yet deleted in BATCH mode
      Check(ClientDist.Retrieve(aStatic.ID[i],V));
      V.YearOfBirth := 1800+nupd;
      Check(ClientDist.BatchUpdate(V)=nupd+n);
      inc(nupd);
    end;
  // add some elements
  V.LastName := 'New';
  for i := 0 to 1000 do
    begin
      V.FirstName := RandomUTF8(10);
      V.YearOfBirth := i+1000;
      Check(ClientDist.BatchAdd(V,true)=n+nupd+i);
    end;
// send the BATCH sequences to the server
Check(ClientDist.BatchSend(Results)=200);
// now all data has been committed on the server
// Results[] contains the results of every BATCH statement...
Check(length(Results)=n+nupd+1001);
// Results[0] to Results[n-1] should be 200 = deletion OK
// Results[n] to Results[n+nupd-1] should be 200 = update OK
// Results[n+nupd] to Results[high(Results)] are the IDs of each added record
for i := 0 to high(Results) do
  if i<nupd+n then
    Check(Results[i]=200) else
    begin
      Check(Results[i]=0);
      ndx := aStatic.IDToIndex(Results[i]);
      Check(ndx>=0);
      with TSQLRecordPeople(aStatic.Items[ndx]) do
        begin
          Check(LastName='New', 'BatchAdd');
          Check(YearOfBirth=1000+i-nupd-n);
        end;
    end;
// check ClientDist.BatchDelete[IntArray[i]] did erase the record
for i := 0 to n-1 do
  Check(not ClientDist.Retrieve(IntArray[i],V),'BatchDelete');
```

In the above code, all CRUD operations are performed as usual, using BatchAdd BatchDelete BatchUpdate methods instead of plain Add Delete Update methods. The ORM will take care of all the
Synopse mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

low-level data process, including JSON serialization, automatic per-chunk transactions creation, and
SQL statements generation, with several optimizations - see below (page 356) and below (page 357).
In the above example, we started the batch process involving only TSQLRecordPeople kind of objects:
Check(ClientDist.BatchStart(TSQLRecordPeople,1000));

But you could mix any kind of TSQLRecord content, if you set the class to nil, as such:
Check(ClientDist.BatchStart(nil,1000));

or use the BatchStartAny() method:
Check(ClientDist.BatchStartAny(1000));

In practice, you should better create and maintain your own instance of TSQLRestBatch, so that you
will be able to implement any number of simultaneous batch process - see below (page 353).

12.3.2. Transmitted JSON
As described above, all Batch*() methods do serialize the objects values as JSON on the client side,
then send this JSON at once to the server, where it will be processed without any client-server
round-trip and slow latency.
Here is some extract of typical JSON stream as sent to the server:
{"People":["DELETE",2,"DELETE",13,"DELETE",24,
(...) all DELETE actions
,"DELETE",11010,
"PUT",{"RowID":3,"FirstName":"Sergei1","LastName":"Rachmaninoff","YearOfBirth":1800,
"YearOfDeath":1943},
"PUT",{"RowID":4,"FirstName":"Alexandre1","LastName":"Dumas","YearOfBirth":1801,
"YearOfDeath":1870},
(...) all PUT = update actions
"PUT",{"RowID":11012,"FirstName":"Leonard","LastName":"da VinÃ§i","YearOfBirth":9025,
"YearOfDeath":1519},
"POST",{"FirstName":"â€š@â€¢Å"Hâ€ mÂ£Â g","LastName":"New","YearOfBirth":1000,
"YearOfDeath":1519},
"POST",{"FirstName":"@â€¦,KAÂ½Ã #Â¶f","LastName":"New","YearOfBirth":1001, "YearOfDeath":1519},
(...) all POST = add actions
"POST",{"FirstName":"+ÂtqCXW3Ã‚\"","LastName":"New","YearOfBirth":2000, "YearOfDeath":1519}
]}

If BatchAdd implies only simple fields (which is the default), those fields name won't be transmitted,
and the following will be emitted in the JSON stream, to reduce needed bandwith:
"SIMPLE",["â€š@â€¢Å"Hâ€ mÂ£Â g","New",1000,1519],

By default, BLOB fields are excluded from the Batch content: only simple fields are send. But
BatchAdd BatchUpdate methods (or corresponding TSQLRestBatch Add or Update methods) could
contain a custom list of fields to be transmitted, in which you could specify any TSQLRawBlob field: the
binary BLOB content will be encoded as Base64 within the JSON process, and you may definitively gain
some resource and speed in such case. Of course, all the data should be small enough to be stored in
memory, so the BLOB fields should better be up to some dozen of MB - use several Batch instances in
a loop, if you have a huge set of data.
On success, the following JSON stream will be received from the server:
[200,200,...]

This array of results is either the HTTP status codes (here 200 means OK), or the inserted new ID (for a
BatchAdd command).
All the JSON generation (client-side) and parsing (server-side) has been optimized to minimize the
SAD - mORMot Framework - Rev. 1.18

Page 353 of 2552


resource needed. With the new internal SynLZ compression (available by default in our HTTP Client-Server classes), used bandwidth is minimal.

Thanks to this BATCH process, most time is now spent into the database engine itself, and not in the communication layer.

### 12.3.3. Unit Of Work pattern

#### 12.3.3.1. Several Batches

On the TSQLRestClientURI side, all BatchStart/BatchAdd/BatchUpdate/BatchDelete methods are using a single temporary storage during the BATCH preparation. This may be safe only if one single thread is accessing the methods - which is usually the case for a REST Client application.

In fact, all BATCH process is using a TSQLRestBatch class, which can be created on the fly, and safely coexist as multiple instances for the same TSQLRest. As a result, you can create your own local TSQLRestBatch instances, for safe batch process. This is in fact mandatory on the TSQLRestServer side, which do not have the Batch*() methods, since they will not be thread safe.

On the server side, you may write for instance:

```pascal
var Batch: TSQLRestBatch;
    IDs: TIntegerDynArray;
...
Batch := TSQLRestBatch.Create(Server,TSQLRecordTest,30);
try
    for i := 10000 to 10099 do begin
        R.Int := i;
        R.Test := Int32ToUTF8(i);
        Check(Batch.Add(R,true)=i-10000);
    end;
    Check(Server.BatchSend(Batch,IDs)=HTTP_SUCCESS);
finally
    Batch.Free;
end;
```

The ability to handle several TSQLRestBatch classes in the same time will allow to implement the Unit Of Work pattern. It can be used to maintain a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems, especially in a complex SOA application with a huge number of connected clients.

In a way, you can think of the Unit of Work as a place to dump all transaction-handling code.

The responsibilities of the Unit of Work are to:

- Manage transactions;
- Order the database inserts, deletes, and updates;
- Prevent concurrency problems;
- Group requests to maximize the database performance.

The value of using a Unit of Work pattern is to free the rest of your code from these concerns so that you can otherwise concentrate on business logic.

#### 12.3.3.2. Updating only the mapped fields

In practice, the BatchUpdate method will only update the mapped fields if called on a record in which a FillPrepare was performed, and not unmapped (i.e. with no call to FillClose). This is required for coherency of the retrieval/modification process.
For instance, in the following code, `V.FillPrepare` will retrieve only ID and YearOfBirth fields of the TSQLRecordPeople table, so subsequent BatchUpdate(V) calls will only update the YearOfBirth field:

```pascal
// test BATCH update from partial FillPrepare
V.FillPrepare(ClientDist,'LastName=:("New")::','ID,YearOfBirth');
if ClientDist.TransactionBegin(TSQLRecordPeople) then
  try
    Check(ClientDist.BatchStart(TSQLRecordPeople));
    n := 0;
    V.LastName := 'NotTransmitted';
    while V.FillOne do begin
      Check(V.LastName='NotTransmitted');
      Check(V.YearOfBirth=n+1000);
      V.YearOfBirth := n;
      ClientDist.BatchUpdate(V); // will update only V.YearOfBirth
      inc(n);
    end;
  end;
(...)```

The transmitted JSON will be computed as such on the client side:

```json
....,"PUT",{"RowID":324,"YearOfBirth":1000},...
```

And the generated SQL on the server side will be:

```sql
UPDATE People SET YearOfBirth=? WHERE RowID=?
... with bound parameters: [1000,324]
```

As a result, BATCH process could be seen as a good way of implementing Unit Of Work for your business layer - see below (page 601). You will be able to modify all your objects as requested, with high-level OOP methods, then have all data transmitted and processed at once when BatchSend() is called. The BatchStart - BatchSend - BatchAbort commands will induce a safe transactional model, relying on the client side for tracking the object modifications, and optimizing the database process on the server side as a simple "save and forget" task, to any SQL or NoSQL engine.

Note that if several ClientDist.BatchUpdate(V) commands are executed within the same FillPrepare() context, they will contain the same fields (RowID and YearOfBirth). They will therefore generate the same statement (UPDATE People SET YearOfBirth=? WHERE RowID=?), which will benefit of Array Binding on the database side - see below (page 356) - if available.

Here is some code, extracted from "web blog" sample "30 - MVC Server", which will update an integer array mapped into a table. All TSQLTag.Occurence integers are stored in a local TSQLTags.Lookup[].Occurence dynamic array, which will be used to display the occurrence count of each tag of the articles.

The following method will first retrieve ID and Occurence from the database, and update the TSQLTag.Occurence if the internal dynamic array contains a new value.

```pascal
procedure TSQLTags.SaveOccurence(aRest: TSQLRest);
var
tag: TSQLTag;
  batch: TSQLRestBatch;
begin
  Lock.ProtectMethod;
  TAutoFree.Several([[@tag,TSQLTag.CreateAndFillPrepare(aRest,'', "RowID,Occurence"),
    @batch,TSQLRestBatch.Create(aRest,TSQLTag,1000)]]);
  while tag.FillOne do begin
    if tag.ID<=length(Lookup) then begin
      if Lookup[tag.ID-1].Occurence<>tag.Occurence then begin
        tag.Occurence := Lookup[tag.ID-1].Occurence;
        batch.Update(tag); // will update only Occurence field
      end;
    end;
```
In the above code, you can identify:
- `CreateAndFillPrepare + FillOne` methods are able to retrieve all values of the `TSQLTag` class, and iterate easily over them;
- A local `TSQLRestBatch` is prepared, and will store locally - via `batch.Update()` - any modification; as we already stated, only the retrieved field (i.e. 'Occurence') will be marked as to be updated;
- `aRest.BatchSend(batch)` will send all new values (if any) to the server, in a single network round trip, and a single transaction;
- This method is made thread safe by using `Lock.ProtectMethod` (Lock is a mutex private to the `TSQLTags` instance);
- Local variables are allocated and automatically released when the method exits, using `TAutoFree.Several()` - see `Automatic TSQLRecord memory handling` (page 149) - which avoid to write two nested `try .. finally Free` end loops.

Such a pattern is very common in mORMot, and illustrate how high-level ORM methods can be used instead of manual SQL. With the potential benefit of a much better performance, and cleaner code.

**12.3.4. Local network as bottleneck**

When using a remote database on a physical network, a round-trip delay occurs for each request, this time between the ORM server side and the external Database engine.

At first, the 1 ms latency due to the external database round-trip may sound negligible. **BATCH sequences for adding/updating/deleting records** (page 350) did already shortcut the Internet latency, which was much higher.
But in a Service-Oriented Architecture (SOA) (page 90), most of the process is done on the server side: the slightest execution delay will induce a noticeable performance penalty. In practice, you won't be able to achieve more than 500-600 requests per second when performing individual INSERT, DELETE or UPDATE statements over any SQL database. Even if run locally on the same server, most SQL databases will suffer from the overhead of inter-process communications, achieving 6,000-7,000 update requests per second at best.

Your customers may not understand why using a SQLite3 engine will be much faster than a dedicated Oracle instance they do pay huge amount of money for, since SQLite3 runs locally in the ORM server process. One common solution is to use stored procedures, or tune the SQL for your database - but you will loose most of the ORM and SOA benefits - see below (page 366).

Of course, mORMot can do better than that. Its ORM will automatically use two ways of diminishing the number of round-trips to the database:
- Using Array Binding - see below (page 356);
- Or multi-INSERT statements - see below (page 357).

Both methods will group all the transmitted data in chunks, as much as possible. Performance will therefore increase, reaching 50,000-60,000 writes per second, depending on the database abilities.

Those features are enabled by default, and the fastest method will always be selected by the ORM core, as soon as it is available on the database back-end. You do not have to worry about configuring your application. Just enjoy its speed.

12.3.4.1. Array binding

12.3.4.1.1. For faster BATCH mode

When used in conjunction with External SQL database access (page 239), BATCH methods can be implemented as array binding if the corresponding TSQLDBConnection class implements the feature. By now, only SynDBOracle, SynDBZeos and SynDBFireDAC units implement it.

Our SynDB.pas unit offers some TSQLDBStatement.BindArray() methods, introducing native array binding for faster database batch modifications. It is working in conjunction with our the BATCH methods of the ORM, so that CRUD modification actions will transparently be grouped within one round-trip over the network.

Thanks to this enhancement, inserting records within Oracle (over a 100 Mb Ethernet network) comes from 400-500 rows per second to more than 70,000 rows per second, according to our Data access benchmark (page 199).

The great maintainers of the ZEOS Open Source library did especially tune its internals to support mORMot at its full speed, directly accessing the ZDBC layer - see ZEOS via direct ZDBC (page 254). The ZEOS 7.2 branch did benefit of a huge code refactoring, and also introduced array binding abilities. This feature will be recognized and handled by our ORM, if available at the ZDBC provider side. Today, only the ZDBC Oracle and Firebird providers do support this feature. But the list is growing.

The FireDAC (formerly AnyDAC) library is the only one implementing this feature (known as Array DML in the FireDAC documentation) around all available Delphi commercial libraries. Enabling it gives a similar performance boost, not only for Oracle, but also MS SQL, Firebird, DB2, MySQL, Informix and PostgreSQL.

In practice, when accessing Oracle, our own direct implementation in SynDBOracle still gives better performance results than the ZDBC / FireDAC implementation.
In fact, some modern database engines (e.g. Oracle or MS SQL) are even faster when using array binding, not only due to the network latency reduce, but to the fact that in such operations, integrity checking and indexes update is performed at the end of the bulk process. If your table has several indexes and constraints, it will make using this feature even faster than a "naive" stored procedure executing individual statements within a loop.

### 12.3.4.1.2. For faster IN clause

Sometimes, you want to write SELECT statements with a huge IN clause. If the number of the items in the IN expression is stable, you may benefit for a prepared statement, e.g.

```sql
SELECT * FROM MyTable WHERE ID IN [?,?,?,?] 
```

But if the IDs are not fixed, you should have to create an expression without any parameter, or use a temporary table:

```sql
SELECT * FROM MyTable WHERE ID IN [1,4,8,12,24,27]
```

As an alternative, SynDBOracle provides the ability to bind an array of parameters which may be cast to an Oracle Object, so that you could use it as a single parameter.

Current implementation support either TInt64DynArray or TRawUTF8DynArray values, as such:

```delphi
var
arr: TInt64DynArray = [1, 2, 3];
Query := TSQLDBOracleConnectionProperties.NewThreadSafeStatementPrepared('select * from table where table.id in '+
  '(select column_value from table(cast(? as SYS.ODCINUMBERLIST)))');
Query.BindArray(1, arr);
Query.ExecutePrepared;
```

RawUTF8 arrays are also supported (which can be used as fall back in case Int64 arrays are not supported by the client, e.g. with Oracle 10):

```delphi
var
  arr: TRawUTF8DynArray = ['123123423452345', '3124234454351324', '53567568578867867'];
Query := TSQLDBOracleConnectionProperties.NewThreadSafeStatementPrepared('select * from table where table.id in '+
  '(select column_value from table(cast(? as SYS.ODCIVARCHAR2LIST)))');
Query.BindArray(1, arr);
Query.ExecutePrepared;
```

From tests on production, this implementation is 2-100 times faster (depending on array and table size) and also simpler, compared to temporary table solution.

Drawback is that it is supported by SynDBOracle only by now.

### 12.3.4.2. Optimized SQL for bulk insert

Sadly, array binding is not available for all databases or libraries.

In order to maximize speed, during BATCH insertion, the mORMot ORM kernel is able to generate some optimized SQL statements, depending on the target database, to send several rows of data at once. It induces a noticeable speed increase when saving several objects into an external database.

Automatic multi-INSERT statement generation is available for:

- Our internal SQLite3 engine (in the mORMotSQLite3.pas unit);
- Almost all the supported External SQL database access (page 239) (in the mORMotDB.pas unit):
  - SQLite3 (3.7.11 and later), MySQL, PostgreSQL, MS SQL Server (2008 and up), Oracle, Firebird, DB2, Informix and NexusDB - and since it is implemented at SQL level, it is available for all supported access libraries, e.g. ODBC, OleDB, Zeos/ZDBC, UniDAC;
- And, in the NoSQL form of "documents array" insertion, for the MongoDB database (in the mORMotMongoDB.pas unit).
It means that even providers not implementing array binding (like OleDB, ODBC or UniDAC) are able to have a huge boost at data insertion.

SQLite3, MySQL, PostgreSQL, MSSQL 2008, DB2, Informix and NexusDB handle INSERT statements with multiple VALUES, in the following SQL-92 standard syntax, using parameters:

```
INSERT INTO TABLE (column-a, [column-b, ...])
VALUES ('value-1a', ['value-1b', ...]),
('value-2a', ['value-2b', ...]),
...
```

Oracle implements the weird-but-similar syntax (note the mandatory SELECT at the end):

```
INSERT ALL
INTO phone_book VALUES ('John Doe', '555-1212')
INTO phone_book VALUES ('Peter Doe', '555-2323')
SELECT * FROM DUAL;
```

Firebird implements its own syntax:

```
execute block
as begin
  INSERT INTO phone_book VALUES ('John Doe', '555-1212');
  INSERT INTO phone_book VALUES ('Peter Doe', '555-2323');
end
```

As a result, most engines show a nice speed boost when using the BatchAdd() method. See Data access benchmark (page 199) for numbers and details.

If you want to use a map/reduce algorithm in your application, or the Unit Of Work pattern (page 353) - in addition to ORM data access - all those enhancements will speed up a lot your data process. Reading and writing huge amount of data has never been so fast and easy: it is time to replace stored-procedure process by high-level code implemented in your Domain service.
12.4. CRUD level cache

Starting with revision 1.16 of the framework, tuned record cache has been implemented at the CRUD/RESTful level, for specific tables or records, on both the server and client sides.

See ORM Cache (page 174) for the other data cache patterns available in the framework, mainly JSON global cache (page 317) at SQLite3 level, on the server side. All mORMot's data caches are using JSON (page 295) as storage format, which was found to be simple and efficient for this purpose.

12.4.1. Where to cache

In fact, a unique caching mechanism is available at the TSQLRest level, for both TSQLRestClient and TSQLRestServer kind of classes. Therefore, Delphi clients can have their own cache, and the Server can also have its own cache. A client without any cache (e.g. a rough AJAX client) will take advantage of the server cache, at least.

By default, there is no caching at REST level. Then you can use the TSQLRest.Cache property to tune your cache policy for each TSQLRest instance.

When caching is set on the server for a particular record or table, in-memory values could be retrieved from this cache instead of calling the database engine each time. When properly used, this will increase global server responsiveness and allow more clients to be served with the same hardware.

On the client side, a local in-memory cache could be first checked when a record is to be retrieved. If the item is found, the client uses this cached value. If the data item is not in the local cache, the query is then sent to the server, just as usual. Due to the high latency of a remote client-server request,
adding caching on the client side does make sense. Client caching properties can be tuned in order to handle properly remote HTTP access via the Internet, which may be much slower than a local Network.

Our caching implementation is transparent to the CRUD code. The very same usual ORM methods are to be called to access a record (Retrieve Update Add), then either client or server cache will be used, if available. For applications that frequently access the same data - a large category - record-level caching improves both performance and scalability.

12.4.2. When to cache

The main problem with cache is about data that both changes and is accessed simultaneously by multiple clients.

In the current implementation, a "pessimistic" concurrency control is used by our framework, relying on explicit locks, and (ab)use of its Stateless ORM (page 314) general design. It is up to the coder to ensure that no major confusion could arise from concurrency issues.

You must tune caching at both Client and Server level - each side will probably require its own set of cache options.

In your project implementation, caching should better not to be used at first, but added on need, when performance and efficiency was found to be required. Adding a cache shall imply having automated regression tests available, since in a Client-Server multi-threaded architecture, "premature optimization is the root of all evil" (Donald Knuth).

The main rules may be simply:
- Not to cache if it may break something relevant (like a global monetary balance value);
- Not to cache unless you need to (see Knuth's wisdom);
- Ensure that caching is worth it (if a value is likely to be overridden often, it could be even slower to cache it);
- Test once, test twice, always test and do not forget to test even more.

In practice, caching issues could be difficult to track. So in case of doubt (why was this data not accurate? it sounds like an old revision?), you may immediately disable caching, then ensure that you were not too optimistic about your cache policy.

12.4.3. What to cache

Typical content of these two tuned caches can be any global configuration settings, or any other kind of unchanging data which is not likely to vary often, and is accessed simultaneously by multiple clients, such as catalog information for an online retailer.

Another good use of caching is to store data that changes but is accessed by only one client at a time. By setting a cache at the client level for such content, the server won't be called often to retrieve the client-specific data. In such case, the problem of handling concurrent access to the cached data doesn't arise.

Profiling can be necessary to identify which data is to be registered within those caches, either at the client and/or the server side. The logging feature - see below (page 641) - integrated to mORMot can be very handy to tune the caching settings, due to its unique customer-side profiling ability.

But most of the time, an human guess at the business logic level is enough to set which data is to be cached on each side, and ensure content coherency.
12.4.4. How to cache

A tuned caching mechanism can be defined, for both TSQLRestClient and TSQLRestServer classes, at ID level.

By default, REST level cache is disabled, until you call TSQLRest.Cache’s SetCache() and SetTimeOut() methods. Those methods will define the caching policy, able to specify which table(s) or record(s) are to be cached, either at the client or the server level.

Once enabled for a table and a set of IDs on a given table, any further call to TSQLRest.Retrieve(aClass,aID) or TSQLRecord.Create(aRest,aID) will first attempt to retrieve the TSQLRecord of the given aID from the internal TSQLRestCache instance's in-memory cache, if available.

Note that more complex requests, like queries on other fields than the ID primary key, or JOINed queries, won't be cached at REST level. But such requests may benefit of the JSON global cache (page 317), at SQLite3 level, on the server side.

For instance, here is how the Client-side caching is tested about one individual record:

```plaintext
(...) Client.Cache.SetCache(TSQLRecordPeople); // cache whole table
TestOne;
Client.Cache.Clear; // reset cache settings
Client.Cache.SetCache(Rec); // cache one record
// same as Client.Cache.SetCache(TSQLRecordPeople,Rec.ID);
TestOne;
(...)
```

In the above code, Client.Cache.Clear is used to reset all cache settings (i.e. not only flush the cache content, but delete all settings previously made with Cache.SetCache() or Cache.SetTimeOut() calls. So in the above code, a global cache is first enabled for the whole TSQLRecordPeople table, then the cache settings are reset, then cache is enabled for only the particular Rec record.

To reset the cache content (e.g. if you consider some values may be deprecated), just call the Cache.Flush methods (able to flush the in-memory cache for all tables, a given table, or a given record).

It's worth warning once again that it's up to the code responsibility to ensure that these caches are consistent over the network. Server side and client side have their own coherency profile to be ensured. The caching policy has to match your data model, and application use cases.

*On the Client side*, only local CRUD operations are tracked. According to the stateless design, adding a time out value does definitively make sense, unless the corresponding data is known to be dedicated to this particular client (like a session data). If no time out period is set, it's up to the client to flush its own cache on purpose, by using TSQLRestClient.Cache.Flush() methods.

*On the Server side*, all CRUD operations of the ORM (like Add / Update / Delete) will be tracked, and cache will be notified of any data change. But direct SQL statements changing table contents (like a UPDATE or a DELETE over one or multiple rows with a WHERE clause) are not tracked by the current implementation: in such case, you'll have to manually flush the server cache content, to enforce data coherency. If such statements did occur on the server side, TSQLRestServer.Cache.Flush() methods are to be called, e.g. in the services which executed the corresponding SQL. If such non-CRUD statements did occur on the client side, it is possible to ensure that the server content is coherent with the client side, via a dedicated TSQLRestClientURI.ServerCacheFlush() method, which will call a dedicated standard service on the server to flush its cache content on purpose.
12.4.5. Business logic and API cache

If your implementation follows a good design - see below (page 602) - the high-level logic is encapsulated into business types, and you won't use directly the TSQLRecord definitions. Another good practice is to define DTO types - see below (page 600) - probably as records or dynamic arrays.

The best performance will be achieved if the data is already known by the service, and returned immediately. Even if our ORM is very fast - thanks to its diverse cache levels we just wrote about - it may be hosted in another service, so a network delay may occur. The less communication, the better.

You may consider using TSynDictionary (page 111) instances over your business objects, or your DTO objects. You may start with no cache in the business or application layers, but once some bottlenecks are identified - e.g. by carefully looking at the logs generated by the framework below (page 641), defining some TSynDictionary instances could help a lot. To release memory, don't forget to setup a proper TimeOutSeconds value.
In your developer background and history, you may have been used to write your business code as stored procedures, to be executed on the server side. In short, a stored procedure is a way of moving some data-intensive SQL process on the database side. A client will ask for some data to be retrieved or processed on the server, and all actions will be taken on the server: since no data has to be exchanged between the client and the server, such a feature is usually much faster than a pure client-sided solution.

Since mORMot is Client/Server from the ground up, it features some unique ways of improving data-intensive process on the client or server sides, without necessary relying on proprietary stored procedures.

This chapter is worth reading, if you start a new mORMot project, and wonder about the architecture of your upcoming applications, or if you are integrating a mORMot server in an existing application... in which you or your predecessors may have (ab)used of stored procedures.

It is time to sit down first, and take counsel how your project may be optimized enough to scale and profit.
13.1. Optimize for performance

So, let's do it the mORMot's way.

As we discussed, the main point about stored procedures is performance. But they are not magic bullet either: we all have seen slow and endless process in stored procedures, almost killing a database server in production. Just as with regular client-side process.

And don't be fooled by performance: make it right, then make it fast.
We could make ourself a motto of this Martin Fowler's remark:

*One of the first questions people consider with this kind of thing is performance. Personally I don't think performance should be the first question. My philosophy is that most of the time you should focus on writing maintainable code. Then use a profiler to identify hot spots and then replace only those hot spots with faster but less clear code. The main reason I do this is because in most systems only a very small proportion of the code is actually performance critical, and it's much easier to improve the performance of well factored maintainable code.*

See http://www.martinfowler.com/articles/dblogic.html - nice link by the way, if you want to identify some best practice about implementing a persistence layer for our business code.

If you are using a mORMot server for the first time, you may be amazed by how most common process will sound just immediate. You can capitalize on the framework optimizations, which are able to unleash the computing power of your hardware, then refine your code only when performance matters.

In order to speed up our data processing, we first have to consider the classic architecture of a mORMot application - see General mORMot architecture - Client Server implementation (page 82). A mORMot client will have two means of accessing its data:
- Either from CRUD / ORM methods;
- Or via services, most probably interface-based services - see below (page 419).

Our optimization goals will therefore be leaded into those two directions.

13.1.1. Profiling your application

If you worry about performance, first reflex may be to enable the framework logging, even on customer sites.

The profiling abilities of the TSynLog class, used everywhere in our framework, will allow you to identify the potential bottlenecks of your client applications. See below (page 631) about this feature.

From our experiment, we can assure you that the first time you setup mORMot advanced logging and profiling on a real application, you may find issues you may never have think of by yourself.

Fight against any duplicated queries, unnecessary returned information (do not forget to specify the fields to be retrieved to your CreateAndFillPrepare() request), unattended requests triggered by the User Interface (typically a TEdit's OnChange event may benefit of using a TTimer before asking for auto-completion)...

Once you have done this cleaning, you may be proud of you, and it may be enough for your customers to enjoy your application. You deserve a coffee or a drink.

13.1.2. Client-side caching
You may have written most of your business logic on the client side, and use CRUD / ORM methods to retrieve the information from your mORMot server. This is perfectly valid, but may be slow, especially if a lot of individual requests are performed over the network, which may be with high latency (e.g. over the Internet).

A new step of optimization, once you did identify your application bottlenecks via profiling, may be to tune your ORM client cache. In fact, there are several layers of caching available in a mORMot application. See CRUD level cache (page 359) for details about these features. Marking some tables as potentially cached on the client side may induce a noticeable performance boost, with no need of changing your client code.

### 13.1.3. Write business logic as Services

A further step of optimization may be to let the business logic be processed on the server side, within a service.

In fact, you will then switch your mind from a classic Multi-tier architecture (page 88) to a Service-Oriented Architecture (SOA) (page 90).

As a result, your process may take much less time, and you may also be able to benefit from some other optimization tricks, like dedicated caching in your service. For instance, consider writing your service in sicShared mode instead of the default sicSingle mode - see below (page 429) - and let some intangible objects be stored as implementation class fields: the next request from a client will not need to load this data from the database, but instead retrieve the information directly from memory, with no latency.

You may also consider using sicClientDriven mode, and cache some client-specific information as implementation class fields.

Beside optimization, your code will probably become easier to maintain and scale, when running as services. SOA is indeed a very convenient pattern, and will induce nice side effects, like the ability to switch to multi-platform clients, including mobile or AJAX, since your business logic will stay on the application server.

### 13.1.4. Using ORM full power

On the server side, your business code, written using CRUD / ORM methods, could be optimized.

First of all, ORM caching may also be used. Any unneeded round-trip to the database - even more with External SQL database access (page 239) - could impact your application responsiveness. Then your business logic, written as services, will benefit from it.

Then you may regroup all your database modifications using BATCH sequences for adding/updating/deleting records (page 350).

This will offer several benefits:
- Transaction support (nothing is written to the database until BatchSend method is executed) similar to the Unit Of Work pattern (page 353);
- Faster insertion, update or deletion - via Array binding (page 356) and Optimized SQL for bulk insert (page 357);
- Perfect integration with the ORM.

In fact, we found out that Array DML or optimized INSERT could be much faster than a regular stored procedure, with individual SQL statements run in a loop.
13.2. Stored procedures

13.2.1. Why to avoid stored procedures

In practice, stored procedures have some huge drawbacks:

- Your business logic is tied to the data layout used for storage - and the Relational Model is far away from natural language - see below (page 596);
- Debugging is somewhat difficult, since stored procedures will be executed on the database server, far away from your application;
- Each developer will probably need its own database instance to be able to debug its own set of stored procedures;
- Project deployment becomes complex, since you have to synchronize your application and database server;
- Cursors and temporary tables, as commonly used in stored procedures, may hurt performance;
- They couple you with a particular database engine: you are tied to use Java, C# or a P/SQL variant to write your business code, then switching from Oracle to PostgreSQL or MS SQL will be error prone, if not impossible;
- They may consume some precious hardware resources on your database server, which may be limited (e.g. proprietary engines like Oracle or MS SQL will force you to use only one CPU or a limited amount of RAM, unless you need to spend a lot of money to increase your license abilities);
- You will probably have limitations in the virtual environment running in your database engine: deprecated VM or libraries, restricted access to files or network due to security requirements, missing libraries;
- Inefficiency of parameters passing, especially when compared with class OOP programming - you are back to the procedural mode of the 80s;
- Parameters passing will probably result in sub-optimal SQL statements, handling all passed values even if not used;
- Flat design of stored procedures interfaces, far away from the interface segregation principle - see below (page 400);
- Let several versions of your business logic coexist on the same server is a nightmare to maintain;
- Unit testing is made difficult, since you won't be able to mock or stub - see below (page 406) - your stored procedures or your data;
- No popular SQL engine does allow stored procedures to be written in Delphi, so you won't be able to share code with your other projects;
- If you use an ORM in your main application, you need to manually maintain the table schema used in your stored procedures in synch with your object model - so you are loosing most of ORM benefits;
- What if you want to switch to NoSQL storage, or a simple stand-alone version of your application?

We do not want to say, dogmatically, that stored procedures are absolute evil. Of course, you are free to use them, even with mORMot.

All we wanted to point out is the fact that they are perhaps not the best fit with the design we would like to follow.

13.2.2. Stored procedures, anyway

There may be some cases where this ORM point of view, may be not enough for your project. Do not worry, as usual mORMot will allow you to do what you need.

The Server-Side services - see below (page 373) and below (page 419) - appear to be the more RESTful compatible way of implementing a stored procedure mechanism in our framework, then consume
them from a mORMot client.

According to the current state of our framework, there are several ways of handling such a server-side SQL/ORM process:
- Write your own SQL function to be used in SQLite3 WHERE statements;
- Low-level dedicated Delphi stored procedures;
- External databases stored procedures.

We will discuss those options.
The first two will in fact implement two types of "stored procedure" at SQL level in pure Delphi code, making our SQLite3 kernel as powerful as other Client-Server RDBMS solutions. The latest option may be considered, especially when moving from legacy applications, still relying on stored procedures for their business logic.

### 13.2.2.1. Custom SQL functions

The SQLite3 engine defines some standard SQL functions, like abs() min() max() or upper(). A complete list is available at [http://www.sqlite.org/lang_corefunc.html](http://www.sqlite.org/lang_corefunc.html).

One of the greatest SQLite3 feature is the ability to define custom SQL functions in high-level language. In fact, its C API allows implementing new functions which may be called within a SQL query. In other database engine, such functions are usually named UDF (for User Defined Functions).

Some custom already defined SQL functions are defined by the framework.
You may have to use, on the Server-side:
- Rank used for page ranking in FTS searches (page 217);
- Concat to process fast string concatenation;
- Soundex SoundexFR SoundexES for computing the English / French / Spanish soundex value of any text;
- IntegerDynArrayContains, ByteDynArrayContains, WordDynArrayContains, CardinalDynArrayContains, Int64DynArrayContains, CurrencyDynArrayContains, RawUTF8DynArrayContainsCase, RawDynArrayContainsNoCase for direct search inside a BLOB column containing some dynamic array binary content (expecting either an INTEGER or a TEXT search value as 2nd parameter).

Those functions are no part of the SQLite3 engine, but are available inside our ORM to handle BLOB containing dynamic array properties, as stated in Dynamic arrays from SQL code (page 161).

Since you may use such SQL functions in an UPDATE or INSERT SQL statement, you may have an easy way of implementing server-side process of complex data, as such:

```sql
UPDATE MyTable SET SomeField=0 WHERE IntegerDynArrayContains(IntArrayField, :(10):)
```

#### 13.2.2.1.1. Implementing a function

Let us implement a CharIndex() SQL function, defined as such:

```sql
CharIndex ( SubText, Text [, StartPos ] )
```

In here, SubText is the string of characters to look for in Text. StartPos indicates the starting index where charindex() should start looking for SubText in Text. Function shall return the position where the match occurred, 0 when no match occurs. Characters are counted from 1, just like in PosEx() Delphi function.

The SQL function implementation pattern itself is explained in the sqlite3.create_function_v2() and TSQLFunctionFunc:
- `argc` is the number of supplied parameters, which are available in `argv[]` array (you can call `ErrorWrongNumberOfArgs(Context)` in case of unexpected incoming number of parameters);
- Use `sqlite3.value_*(argv[*])` functions to retrieve a parameter value;
- Then set the result value using `sqlite3.result_*` functions.

Here is typical implementation code of the `CharIndex()` SQL function, calling the expected low-level SQLite3 API (note the cdecl calling convention, since it is a SQLite3 / C callback function):

```pascal
procedure InternalSQLFunctionCharIndex(Context: TSQLite3FunctionContext; argc: integer; var argv: TSQLite3ValueArray); cdecl;
var
  StartPos: integer;
begin
  case argc of
    2: StartPos := 1;
    3:
      begin
        StartPos := sqlite3.value_int64(argv[2]);
        if StartPos<=0 then
          StartPos := 1;
      end;
    else
      begin
        ErrorWrongNumberOfArgs(Context);
        exit;
      end;
  end;
  if (sqlite3.value_type(argv[0])=SQLITE_NULL) or
     (sqlite3.value_type(argv[1])=SQLITE_NULL) then
    sqlite3.result_int64(Context,SynCommons.PosEx(
      sqlite3.value_text(argv[0]),sqlite3.value_text(argv[1]),StartPos));
end;
```

This code just get the parameters values using `sqlite3.value_*()` functions, then call the `PosEx()` function to return the position of the supplied text, as an INTEGER, using `sqlite3.result_int64()`.

The local `StartPos` variable is used to check for an optional third parameter to the SQL function, to specify the character index to start searching from.

The special case of a NULL parameter is handled by checking the incoming argument type, calling `sqlite3.value_type(argv[])`.

### 13.2.2.1.2. Registering a function

#### 13.2.2.1.2.1. Direct low-level SQLite3 registration

Since we have a `InternalSQLFunctionCharIndex()` function defined, we may register it with direct SQLite3 API calls, as such:

```pascal
sqlite3.create_function_v2(Demo.DB, 'CharIndex', 2, SQLITE_ANY, nil, InternalSQLFunctionCharIndex, nil, nil, nil);
sqlite3.create_function_v2(Demo.DB, 'CharIndex', 3, SQLITE_ANY, nil, InternalSQLFunctionCharIndex, nil, nil, nil);
```

The function is registered twice, one time with 2 parameters, then with 3 parameters, to add an overloaded version with the optional `StartPos` parameter.

#### 13.2.2.1.2.2. Class-driven registration

It is possible to add some custom SQL functions to the SQLite3 engine itself, by creating a `TSQLDataBaseSQLFunction` custom class and calling the `TSQLDataBase.RegisterSQLFunction` method.
The standard way of using this is to override the TSQLRestServerDB.InitializeEngine virtual method, calling DB.RegisterSQLFunction() with an defined TSQLDataBaseSQLFunction custom class.

So instead of calling low-level sqlite3.create_function_v2() API, you can declare the CharIndex SQL function as such:

```pascal
Demo.RegisterSQLFunction(InternalSQLFunctionCharIndex, 2, 'CharIndex');
Demo.RegisterSQLFunction(InternalSQLFunctionCharIndex, 3, 'CharIndex');
```

The two lines above will indeed wrap the following code:

```pascal
Demo.RegisterSQLFunction(TSQLDataBaseSQLFunction.Create(InternalSQLFunctionCharIndex, 2, 'CharIndex'));
Demo.RegisterSQLFunction(TSQLDataBaseSQLFunction.Create(InternalSQLFunctionCharIndex, 3, 'CharIndex'));
```

The RegisterSQLFunction() method is called twice, one time with 2 parameters, then with 3 parameters, to add an overloaded version with the optional StartPos parameter, as expected.

### 13.2.2.1.2.3. Custom class definition

The generic function definition may be completed, in our framework, with a custom class definition, which is handy to have some specific context, not only relative to the current SQL function context, but global and static to the whole application process.

![TSQLDataBaseSQLFunction classes hierarchy](image)

For instance, the following method will register a SQL function able to search into a BLOB-stored custom dynamic array type:

```pascal
procedure TSQLDataBase.RegisterSQLFunction(aDynArrayTypeInfo: pointer; 
aCompare: TDynArraySortCompare; const aFunctionName: RawUTF8); 
begin 
  RegisterSQLFunction( 
    TSQLDataBaseSQLFunctionDynArray.Create(aDynArrayTypeInfo,aCompare,aFunctionName));
end;
```

We specify directly the TSQLDataBaseSQLFunctionDynArray class instance to work with, which adds two needed protected fields to the TSQLDataBaseSQLFunction root class:

- A `fDummyDynArray` TDynArray instance which will handle the dynamic array RTTI handling;
- A `fDummyDynArrayValue` pointer, to be used to store the dynamic array reference values to be used during the dynamic array process.

Here is the corresponding class definition:

```pascal
/// to be used to define custom SQL functions for dynamic arrays BLOB search
TSQLDataBaseSQLFunctionDynArray = class(TSQLDataBaseSQLFunction)
```
protected
    fDummyDynArray: TDynArray;
fDummyDynArrayValue: pointer;

public
    /// Initialize the corresponding SQL function
    /// - if the function name is not specified, it will be retrieved from the type
    /// // information (e.g. TReferenceDynArray will declare 'ReferenceDynArray')
    /// // - the SQL function will expect two parameters: the first is the BLOB
    /// // field content, and the 2nd is the array element to search (set with
    /// // TDynArray.ElemSave() or with BinToBase64WithMagic(aDynArray.ElemSave())
    /// // if called via a Client and a JSON prepared parameter)
    /// // - you should better use the already existing faster SQL functions
    /// // Byte/Word/Integer/Cardinal/Int64/CurrencyDynArrayContains() if possible
    /// // (this implementation will allocate each dynamic array into memory before
    /// // comparison, and will be therefore slower than those optimized versions)
    constructor Create(aTypeInfo: pointer; aCompare: TDynArraySortCompare;
                      const aFunctionName: RawUTF8=''); override;
end;

And the constructor implementation:

constructor TSQLDataBaseSQLFunctionDynArray.Create(aTypeInfo: pointer;
                                                   aCompare: TDynArraySortCompare;
                                                   const aFunctionName: RawUTF8);
begind
    fDummyDynArray.Init(aTypeInfo,fDummyDynArrayValue);
    fDummyDynArray.Compare := aCompare;
    inherited Create(InternalSQLFunctionDynArrayBlob,2,aFunctionName);
end;

The InternalSQLFunctionDynArrayBlob function is a low-level SQLite3 engine SQL function
prototype, which will retrieve a BLOB content, then un-serialize it into a dynamic array (using the
fDummyDynArrayValue. LoadFrom method), then call the standard ElemLoadFind method to search
the supplied element, as such:

(...)
with Func.fDummyDynArray do
try
    LoadFrom(DynArray); // temporary allocate all dynamic array content
try
    if ElemLoadFind(Elem)<0 then
        DynArray := nil;
finally
    Clear; // release temporary array content in fDummyDynArrayValue
end;
(...)

You can define a similar class in order to implement your own custom SQL function.

Here is how a custom SQL function using this TSQLDataBaseSQLFunctionDynArray class is registered
in the supplied unitary tests to an existing database connection:

Demo.RegisterSQLFunction(TypeInfo(TIntegerDynArray),SortDynArrayInteger,
                           'MyIntegerDynArrayContains');

This new SQL function expects two BLOBs arguments, the first being a reference to the BLOB column,
and the 2nd the searched value. The function can be called as such (lines extracted from the
framework regression tests):

aClient.OneFieldValues(TSQLRecordPeopleArray,'ID',
                       FormatUTF8('MyIntegerDynArrayContains(?,%):',
                       [BinToBase64WithMagic(@k,sizeof(k))]),IDs);

Note that since the 2nd parameter is expected to be a BLOB representation of the searched value, the
BinToBase64WithMagic function is used to create a BLOB parameter, as expected by the ORM. Here,
the element type is an integer, which is a pure binary variable (containing no reference-counted
internal fields): so we use direct mapping from its binary in-memory representation; for more complex
element type, you should use the generic BinToBase64WithMagic(aDynArray.ElemSave()) expression instead, calling TDynArray. ElemSave method.

Note that we did not use here the overloaded OneFieldValues method expecting '?' bound parameters here, but we may have use it as such:

```delphi
acClient.OneFieldValues(TSQLRecordPeopleArray,'ID',
  FormatUTF8('MyIntegerDynArrayContains(Ints,?)',[],
  [BinToBase64WithMagic(@k,sizeof(k))],IDs);
```

Since the MyIntegerDynArrayContains function will create a temporary dynamic array in memory from each row (stored in fDummyDynArrayValue), the dedicated IntegerDynArrayContains SQL function is faster.

### 13.2.2.2. Low-level SQLite3 stored procedure in Delphi

To implement a more complete request, and handle any kind of stored data in a column (for instance, some TEXT format to be parsed), a TOnSQLStoredProc event handler can be called for every row of a prepared statement, and is able to access directly to the database request. This event handler can be specified to the TSQLRestServerDB.StoredProcExecute() method. Be aware that code inside this event handler should not use the ORM methods of the framework, but direct low-level SQLite3 access (to avoid re-entrance issues).

This will allow direct content modification during the SELECT statement. Be aware that, up to now, Virtual Tables magic (page 225) TSQLVirtualTableCursorJSON cursors are not safe to be used if the Virtual Table data is modified.

See the description of the TOnSQLStoredProc event handler and associated StoredProcExecute() method in the second part of this document.

### 13.2.2.3. External stored procedure

If the application relies on external databases - see External SQL database access (page 239) - the external database may be located on a remote computer.

In such situation, all RESTful Server-sided solutions could produce a lot of network traffic. In fact, custom SQL functions or stored procedures both use the SQLite3 engine as root component.

In order to speed up the process, you may define some RDMS stored procedures in the external database syntax (P/SQL, .Net, Java or whatever), then define some below (page 372) to launch those functions.

Note that in this case, you'll loose the database independence of the framework, and most of the benefits of using an ORM/ODM - later on, switching to another database engine may become impossible. Such RDBMS stored procedures may be envisaged only during the transition phase of an existing application. BATCH sequences for adding/updating/deleting records (page 350) has almost all the speed advantages of stored procedures, with the benefit of a pure object oriented code, easy to debug and maintain.
13.3. Server side Services

In order to follow a Service-Oriented Architecture (SOA) (page 90) design, your application’s business logic can be implemented in several ways using mORMot:

- Via some TSQLRecord inherited classes, inserted into the database model, and accessible via some RESTful URI - this is implemented by our ORM architecture - see Client-Server process (page 318);
- By some RESTful services, implemented in the Server as published methods, and consumed in the Client via native Delphi methods;
- Defining some RESTful service contracts as standard Delphi interface, and then run it seamlessly on both client and client sides.

The first is similar to RemObject’s DataAbstract product, which allows remote access to database, over several protocols. There are some similarities with mORMot (like on-the-fly SQL translation for external databases), but also a whole diverse use case (RAD/components and wizards versus ORM/MVC) and implementation (mORMot takes advantages of the SQLite3 SQL core and is much more optimized for speed and scaling).

If you paid for a Delphi Architect edition, the first two items can be compared to the DataSnap Client-Server features. Since Delphi 2010, you can in fact define JSON-based RESTful services, in addition to the original DCOM/DBExpress remote data broker. It makes uses of the new RTTI available since Delphi 2010, but it has some known stability and performance issues, and lack of strong security. It is also RAD/Wizard based, whereas mORMot uses a code approach.

The last item is purely interface-based, so matches the "designed by contract" principle - see below (page 389) - as implemented by Microsoft’s WCF technology - see below (page 478). We included most of the nice features made available in WCF in mORMot, in a KISS convention over configuration manner.

So mORMot is quite unique, in the fact that it features, in one unique code base, all three ways of implementing a SOA application. And it is an Open Source project, existing since years - you won’t be stucked with proprietary code nor licenses. You can move your existing code base into a Domain-Driven Design, on your management pace (and money), without the need of upgrading to the latest version of the IDE.
14. Client-Server services via methods

To implement a service in the *Synopse mORMot framework*, the first method is to define published method Server-side, then use easy functions about JSON or URL-parameters to get the request encoded and decoded as expected, on Client-side.

We’ll implement the same example as in the official Embarcadero docwiki page above. Add two numbers. Very useful service, isn't it?

### 14.1. Publishing a service on the server

On the server side, we need to customize the standard TSQLRestServer class definition (more precisely a TSQLRestServerDB class which includes a SQLite3 engine, or a lighter TSQLRestServerFullMemory kind of server, which is enough for our purpose), by adding a new published method:

```pascal
type
TSQLRestServerTest = class(TSQLRestServerFullMemory)
  (...) 
  published
  procedure Sum(Ctxt: TSQLRestServerURIContext);
end;
```

The method name ("Sum") will be used for the URI encoding, and will be called remotely from `ModelRoot/Sum` URL. The `ModelRoot` is the one defined in the Root parameter of the *model* used by the application.

This method, like all Server-side methods, MUST have the same exact parameter definition as in the TSQLRestServerCallback prototype, i.e. only one Ctxt parameter, which refers to the whole
execution context:

```plaintext
type
  TSQLRestServerCallBack = procedure(Ctxt: TSQLRestServerURIContext) of object;

Then we implement this method:

```plaintext
procedure TSQLRestServerTest.Sum(Ctxt: TSQLRestServerURIContext);
begin
  Ctxt.Results([Ctxt['a']+Ctxt['b']]);
end;
```

The Ctxt variable publish some properties named InputInt[], InputDouble[], InputUTF8[] and Input[] able to retrieve directly a parameter value from its name, respectively as Integer/Int64, double, RawUTF8 or variant. The Ctxt.Input[] array property, returning variant property for the TSQLRestServerURIContext class, so writing Ctxt['a'] is the same as writing Ctxt.Input['a'].

Therefore, the code above using Ctxt[] or Ctxt.Input[] will introduce a conversion via a variant, which may be a bit slower, and in case of string content, may lose some content for older non Unicode versions of Delphi. So it is a good idea to use the exact expected Input*[] property corresponding to your value type. It does make sense even more when handling text, i.e. InputUTF8[] is to be used in such case. For our floating-point computation method, we may have coded it as such:

```plaintext
procedure TSQLRestServerTest.Sum(Ctxt: TSQLRestServerURIContext);
begin
  with Ctxt do
    Results([InputDouble['a']+InputDouble['b']]);
end;
```

The Ctxt.Results([]) method is used to return the service value as one JSON object with one "Result" member, with default MIME-type JSON_CONTENT_TYPE.

For instance, the following request URI:

```plaintext
GET /root/Sum?a=3.12&b=4.2
```

will let our server method return the following JSON object:

```plaintext
{"Result":7.32}
```

That is, a perfectly AJAX-friendly request.

Note that all parameters are expected to be plain case-insensitive 'A'..'Z','0'..'9' characters.

An important point is to remember that the implementation of the callback method must be thread-safe - as stated by Thread-safety (page 335) and Safe locks for multi-thread applications (page 123). In fact, the TSQLRestServer.URI method expects such callbacks to handle the thread-safety on their side. It's perhaps some more work to handle a critical section in the implementation, but, in practice, it's the best way to achieve performance and scalability: the resource locking can be made at the tiniest code level.
14.2. Defining the client

The client-side is implemented by calling some dedicated methods, and providing the service name ("sum") and its associated parameters:

```pascal
function Sum(aClient: TSQLRestClientURI; a, b: double): double;
var err: integer;
begin
  val(aClient.CallBackGetResult('sum', ['a', a, 'b', b]), Result, err);
end;
```

You could even implement this method in a dedicated client method - which make sense:

```pascal
type
  TMyClient = class(TSQLHttpClient) // could be TSQLRestClientURINamedPipe
    (...) end;

function TMyClient.Sum(a, b: double): double;
var err: integer;
begin
  val(CallBackGetResult('sum', ['a', a, 'b', b]), Result, err);
end;
```

This later implementation is to be preferred on real applications.

You have to create the server instance, and the corresponding TSQLRestClientURI (or TMyClient), with the same database model, just as usual...

On the Client side, you can use the CallBackGetResult method to call the service from its name and its expected parameters, or create your own caller using the UrlEncode() function. Note that you can specify most class instance into its JSON representation by using some TObject into the method arguments:

```pascal
function TMyClient.SumMyObject(a, b: TMyObject): double;
var err: integer;
begin
  val(CallBackGetResult('summyobject', ['a', a, 'b', b]), Result, err);
end;
```

This Client-Server protocol uses JSON here, as encoded server-side via Ctxt.Results() method, but you can serve any kind of data, binary, HTML, whatever... just by overriding the content type on the server with Ctxt.Returns().
14.3. Direct parameter marshalling on server side

We have used above the Ctxt[] and Ctxt.Input*[] properties to retrieve the input parameters. This is pretty easy to use and powerful, but the supplied Ctxt gives full access to the input and output context.

Here is how we may implement the fastest possible parameters parsing - see sample Project06Server.dpr:

```pascal
procedure TSQLRestServerTest.Sum(Ctxt: TSQLRestServerURIContext);
var a,b: double;
begin
  if UrlDecodeNeedParameters(Ctxt.Parameters, 'A,B') then begin
    while Ctxt.Parameters<>nil do begin
      UrlDecodeDouble(Ctxt.Parameters, 'A=', a);
      UrlDecodeDouble(Ctxt.Parameters, 'B=', b, @Ctxt.Parameters);
    end;
    Ctxt.Results([a+b]);
  end else
    Ctxt.Error('Missing Parameter');
end;
```

The only not obvious part of this code is the parameters marshalling, i.e. how the values are retrieved from the incoming Ctxt.Parameters text buffer, then converted into native local variables.

On the Server side, typical implementation steps are therefore:
- Use the UrlDecodeNeedParameters function to check that all expected parameters were supplied by the caller in Ctxt.Parameters;
- Call UrlDecodeInteger / UrlDecodeInt64 / UrlDecodeDouble / UrlDecodeExtended / UrlDecodeValue / UrlDecodeObject functions (all defined in SynCommons.pas) to retrieve each individual parameter from standard JSON content;
- Implement the service (here it is just the a+b expression);
- Then return the result calling Ctxt.Results() method or Ctxt.Error() in case of any error.

The powerful UrlDecodeObject function (defined in mORMot.pas) can be used to un-serialize most class instance from its textual JSON representation (TPersistent, TSQLRecord, TStringList...).

Using Ctxt.Results() will encode the specified values as a JSON object with one "Result" member, with default mime-type JSON_CONTENT_TYPE:

```json
{"Result":"OneValue"}
```

or a JSON object containing an array:

```json
{"Result":["One","two"]}
```
14.4. Returns non-JSON content

Using Ctxt.Returns() will let the method return the content in any format, e.g. as a JSON object (via the overloaded Ctxt.Returns([]) method expecting field name/value pairs), or any content, since the returned MIME-type can be defined as a parameter to Ctxt.Returns() - it may be useful to specify another mime-type than the default constant JSON_CONTENT_TYPE, i.e. 'application/json; charset=UTF-8', and returns plain text, HTML or binary.

For instance, you can return directly a value as plain text:

```delphi
procedure TSQLRestServer.Timestamp(Ctxt: TSQLRestServerURIContext);
begin
  Ctxt.Returns(Int64ToUtf8(ServerTimestamp), HTTP_SUCCESS, TEXT_CONTENT_TYPE_HEADER);
end;
```

Or you can return some binary file, retrieving the corresponding MIME type from its binary content:

```delphi
procedure TSQLRestServer.GetFile(Ctxt: TSQLRestServerURIContext);
var
  fileName: TFileName;
  content: RawByteString;
  contentType: RawUTF8;
begin
  fileName := 'c:\data\'+ExtractFileName(Ctxt['filename']); // or Ctxt.Input['filename']
  content := StringFromFile(fileName);
  if content=''
    then
    Ctxt.Error('',HTTP_NOTFOUND)
  else
    Ctxt.Returns(content,HTTP_SUCCESS,HEADER_CONTENT_TYPE+
      GetMimeContentType(pointer(content),Length(content),fileName));
end;
```

The corresponding client method may be defined as such:

```delphi
function TMyClient.GetFile(const aFileName: RawUTF8): RawByteString;
begin
  if CallbackGet('GetFile',['filename'],aFileName,RawUTF8(result):HTTP_SUCCESS then
    raise Exception.CreateFmt('Impossible to get file: %s',[result]);
end;
```

Note that the Ctxt.ReturnFile() method - see below (page 379) - is preferred than manual file retrieval as implemented in this TSQLRestServer.GetFile() method. It is shown here for demonstration purposes only.

If you use HTTP as communication protocol, you can consume these services, implemented Server-Side in fast Delphi code, with any AJAX application on the client side.

Using GetMimeContentType() when sending non JSON content (e.g. picture, pdf file, binary...) will be interpreted as expected by any standard Internet browser: it could be used to serve some good old HTML content within a page, not necessary consume the service via JavaScript.
14.5. Advanced process on server side

On server side, method definition has only one Ctxt parameter, which has several members at calling time, and publish all service calling features and context, including RESTful URI routing, session handling or low-level HTTP headers (if any).

At first, Ctxt may indicate the expected TSQLRecord ID and TSQLRecord class, as decoded from RESTful URI. It means that a service can be related to any table/class of our ORM framework, so you will be able to create easily any RESTful compatible requests on URI like ModelRoot/TableName/TableID/MethodName. The ID of the corresponding record is decoded from its RESTful scheme into Ctxt.TableID, and the table is available in Ctxt.Table or Ctxt.TableIndex (if you need its index in the associated server Model).

For example, here we return a BLOB field content as hexadecimal, according to its TableName/TableID:

```pascal
procedure TSQLRestServerTest.DataAsHex(Ctxt: TSQLRestServerURIContext);
var
  aData: TSQLRawBlob;
begin
  if (self=nil) or (Ctxt.Table<>TSQLRecordPeople) or (Ctxt.TableID<=0) then
    Ctxt.Error('Need a valid record and its ID')
  else
    if RetrieveBlob(TSQLRecordPeople,Ctxt.TableID,'Data',aData) then
      Ctxt.Results([SynCommons.BinToHex(aData)])
    else
      Ctxt.Error('Impossible to retrieve the Data BLOB field');
end;
```

A corresponding client method may be:

```pascal
function TSQLRecordPeople.DataAsHex(aClient: TSQLRestClientURI): RawUTF8;
begin
  Result := aClient.CallBackGetResult('DataAsHex',[],RecordClass,fID);
end;
```

If authentication - see below (page 546) - is used, the current session, user and group IDs are available in Session / SessionUser / SessionGroup fields. If authentication is not available, those fields are meaningless: in fact, Ctxt.Context.Session will contain either 0 (CONST_AUTHENTICATION_SESSION_NOT_STARTED) if any session is not yet started, or 1 (CONST_AUTHENTICATION_NOT_USED) if authentication mode is not active. Server-side implementation can use the TSQLRestServer.SessionGetUser method to retrieve the corresponding user details (note that when using this method, the returned TSQLAuthUser instance is a local thread-safe copy which shall be freed when done).

In Ctxt.Call^ member, you can access low-level communication content, i.e. all incoming and outgoing values, including headers and message body. Depending on the transmission protocol used, you can retrieve e.g. HTTP header information. For instance, here is how you may access the client remote IP address and application User-Agent, at lowest level:

```pascal
aRemoteIP := FindIniNameValue(pointer(Ctxx.Call.InHead), 'REMOTEIP: ');
aUserAgent := FindIniNameValue(pointer(Ctxx.Call.InHead), 'USER-AGENT: ');
```

Of course, for those fields, it is much preferred to use the Ctxt.RemoteIP or Ctxt.UserAgent properties, which use an efficient cache.
14.6. Browser speed-up for unmodified requests

When used over a slow network (e.g. over the Internet), you can set the optional Handle304NotModified parameter of both Ctxt.Returns() and Ctxt.Results() methods to return the response body only if it has changed since last time.

In practice, result content will be hashed (using crc32c algorithm, and fast SSE4.2 hardware instruction, if available) and in case of no modification will return "304 Not Modified" status to the browser, without the actual result content. Therefore, the response will be transmitted and received much faster, and will save a lot of bandwidth, especially in case of periodic server pooling (e.g. for client screen refresh).

Note that in case of hash collision of the crc32c algorithm (we never did see it happen, but such a mathematical possibility exists), a false positive "not modified" status may be returned; this option is therefore unset by default, and should be enabled only if your client does not handle any sensitive accounting process, for instance.

Be aware that you should disable authentication for the methods using this Handle304NotModified parameter, via a TSQLRestServer.ServiceMethodByPassAuthentication() call. In fact, our RESTful authentication - see below (page 546) - uses a per-URI signature, which change very often (to avoid men-in-the-middle attacks). Therefore, any browser-side caching benefit will be voided if authentication is used: browser internal cache will tend to grow for nothing since the previous URIs are deprecated, and it will be a cache-miss most of the time. But when serving some static content (e.g. HTML content, fixed JSON values or even UI binaries), this browser-side caching can be very useful.

This stateless REST (page 311) model will enable several levels of caching, even using an external Content Delivery Network (CDN) service. See below (page 542) for some potential hosting architectures, which may let your mORMot server scale to thousands of concurrent users, served around the world with the best responsiveness.

14.7. Returning file content

Framework's HTTP server is able to handle returning a file as response to a method-based service. The High-performance http.sys server (page 326) is even able to serve the file content asynchronously from kernel mode, with outstanding performance.

You can use the Ctxt.ReturnFile() method to return a file directly. This method is also able to guess the MIME type from the file extension, and handle HTTP NOTMODIFIED = 304 process, if Handle304NotModified parameter is true, using the file time stamp.

Another possibility may be to use the Ctxt.ReturnFileFromFolder() method, which is able to efficiently return any file specified by its URI, from a local folder. It may be very handy to return some static web content from a mORMot HTTP server.

14.8. JSON Web Tokens (JWT)

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC
algorithm) or a public/private key pair using RSA or ECDSA. They can be used for:

- Authentication: including a JWT to any HTTP request allows Single Sign On user validation across different domains;
- Secure Information Exchange: a small amount of data can be stored in the JWT payload, and is digitally signed to ensure its provenance and integrity.

See [http://jwt.io](http://jwt.io) for an introduction to JSON Web Tokens.

Our framework implements JWT:

- "HS256/384/512" (HMAC-SHA2-256/384/512), "ES256" (256-bit ECDSA) standard algorithms, and "S3256/384/512" (for non-yet-standard SHA3-256/384/512) - with the addition of the "none" weak algo, to be used with caution;
- Computes and validates all JWT claims: dates, audiences, JWT ID;
- Thread-safe and high performance (2 us for a HS256 verification under x64), with optional in-memory cache if needed (e.g. for slower ES256);
- Stand-alone and cross-platform code: no external dll, works with Delphi or FPC;
- Enhanced security - it is by design immune from [https://auth0.com/blog/2015/03/31/critical-vulnerabilities-in-json-web-token-libraries](https://auth0.com/blog/2015/03/31/critical-vulnerabilities-in-json-web-token-libraries);
- Full integration with the framework.

It is architectured around a set of classes, one per algorithm, following the least astonishment principle, and enhancing security:

![TJWTAbstract classes hierarchy](image)

In SynCrypto.pas and SynEcc.pas, you will find:

- TJWTAbstract as abstract parent class for implementing JSON Web Tokens;
- TJWTNone implementing the "none" algorithm;
- TJWTHS256 TJWTHS384 TJWTHS512 implementing the "HS256 HS384 HS512" algorithms, i.e. HMAC-SHA2 over 256, 384 or 512 bits;
- TJWTS3256 TJWTS3384 TJWTS3512 implementing the "S3256 S3384 S3512" algorithms, i.e. SHA3 over 256, 384 or 512 bits;
- TJWTES256 implementing the "ES256" algorithm, i.e. ECDSA using the P-256 curve and the SHA-256
hash algorithm.

To work with JWT, you may write for instance:

```pascal
var j: TJWTAbstract;
jwt: TJWTContent;
...
j := TJWTHS256.Create('secret',0,[jrcSubject],[]);
try
    j.Verify('eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA950rM7E2cBab3ORMHpHdCExj0Y7gE+','ONFh7HgQ',jwt); // reference from jwt.io
    check(jwt.result=jwtValid);
    check(jwt.reg[jrcSubject]=1234567890);
    check(jwt.data.U['name']=John Doe);
    check(jwt.data.B['admin']);
finally
    j.Free;
end;
```

The 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...' token contains in fact the following, once base-64 decoded:

- header: {
  "alg":"HS256","typ":"JWT"
- payload: {
  "sub":"1234567890","name":"John Doe","admin":true
- signature: HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), "secret")

The TJWTAbstract classes implement the logic from supplied security parameters about a given set of JWT, then you can use TJWTAbstract.Verify to decode and check the payload and signature of a JWT into a TJWTContent local variable. As you can see, TJWTContent.result contains the decoding status, TJWTContent.reg[] the decoded claims, and TJWTContent.data is a TDocVariant custom variant type (page 112) giving access to any stored private information.

It has a built-in support of JWT claims when tokens are generated, so you can write:

```pascal
j := TJWTHS256.Create('sec',10,[jrcIssuer,jrcExpirationTime,jrcIssuedAt,jrcJWTID],[],60);
token := j.Compute(['http://example.com/is_root',true], 'joe');
```

Now, the token variable contains e.g. as signed payload:

```pascal
{"http://example.com/is_root":true,"iss":"joe","iat":1482177879,"exp":1482181479,"jti":"1496DCE06769250338B5A81"}
```

The issuer has been encoded as an expected "iss": field, "iat" and "exp" fields contain the issuing and expiration timestamps, and "jti" has been filled with an obfuscated TSynUniqueIdentifier as JWT ID. Since we use a TJWTHS256 class, HMAC-SHA256 digital signature of the header and payload has then been appended - with a secret safely derivated from 'sec' passphrase using 10 rounds of a PBKDF2_SHA256 derivation (in practice, you may use a much higher number like 20,000).

Then you can decode such a token, and access its payload in a single method:

```pascal
j.Verify(token,jwt);
assert(jwt.result=jwtValid);
assert(jwt.reg[jrcIssuer]='joe');
```

Integration with method-based services is easy, using TSQLRestServerURIContext.AuthenticationCheck method:
Ctxt.ReturnFileFromFolder('c:\datafolder');
end;

The above method will define a method-based service returning the content of a local folder, only if a valid JWT is supplied within the HTTP headers of the incoming request. If AuthenticationCheck fails to validate the token supplied in the associated Ctxt, it will return 401 HTTP_UNAUTHORIZED to the client, as expected.

An alternative to use JWT for authentication may be to assign a TJWTAbs tract inherited instance to TSQLRestServer.JWTForUnauthenticatedRequest - see below (page 557).
14.9. Handling errors

When using Ctxt.Input*[] properties, any missing parameter will raise an EParsingException. It will therefore be intercepted by the server process (as any other exception), and returned to the client with an error message containing the Exception class name and its associated message.

But you can have full access to the error workflow, if needed. In fact, calling either Ctxt.Results(), Ctxt.Returns() or Ctxt.Error() will specify the HTTP status code (e.g. 200 / "OK" for Results() and Success() methods by default, or 400 / "Bad Request" for Error()) as an integer value. For instance, here is how a service not returning any content can handle those status/error codes:

```pascal
procedure TSQLRestServer.Batch(Ctxt: TSQLRestServerURIContext);
begin
  if (Ctxt.Method=mPUT) and RunBatch(nil, nil, Ctxt) then
    Ctxt.Success
  else
    Ctxt.Error;
end;
```

In case of an error on the server side, you may call Ctxt.Error() method (only the two valid status codes are 200 and 201).

The Ctxt.Error() method has an optional parameter to specify a custom error message in plain English, which will be returned to the client in case of an invalid status code. If no custom text is specified, the framework will return the corresponding generic HTTP status text (e.g. "Bad Request" for default status code HTTP_BADREQUEST = 400).

In this case, the client will receive a corresponding serialized JSON error object, e.g. for Ctxt.Error('Missing Parameter',HTTP_NOTFOUND):

```json
{
  "ErrorCode":404,
  "ErrorText":"Missing Parameter"
}
```

If called from an AJAX client, or a browser, this content should be easy to interpret.

Note that the framework core will catch any exception during the method execution, and will return a "Internal Server Error" / HTTP_SERVERERROR = 500 error code with the associated textual exception details.
14.10. Benefits and limitations of this implementation

Method-based services allow fast and direct access to all mORMot Client-Server RESTful features, over all usual protocols of our framework: HTTP/1.1, Named Pipe, Windows Messages, direct in-memory/in-process access.

The mORMot implementation of method-based services gives full access to the lowest-level of the framework core, so it has some advantages:

- It can be tuned to fit any purpose (such as retrieving or returning some HTML or binary data, or modifying the HTTP headers on the fly);
- It is integrated into the RESTful URI model, so it can be related to any table/class of our ORM framework (like DataAsHex service above), or it can handle any remote query (e.g. any AJAX or SOAP requests);
- It has a very low performance overhead, so can be used to reduce server workload for some common tasks.

Note that due to this implementation pattern, the mORMot service implementation is very fast, and not sensitive to the "Hash collision attack" security issue, as reported with Apache - see http://blog.synopse.info/post/2011/12/30/Hash-collision-attack for details.

But with this implementation, a lot of process (e.g. parameter marshalling) is to be done by hand on both client and server side code. In addition, building and maintaining a huge SOA system with a "method by method" approach could be difficult, since it publishes one big "flat" set of services. This is where interfaces enter the scene.
15. Interfaces

15.1. Delphi and interfaces

15.1.1. Declaring an interface

No, interface(-book) is not another social network, sorry.

In Delphi OOP model, an interface defines a type that comprises abstract virtual methods. The short, easy definition is that an interface is a declaration of functionality without an implementation of that functionality. It defines "what" is available, not "how" it is made available. This is the so called "abstraction" benefit of interfaces (there are another benefits, like orthogonality of interfaces to classes, but we'll see it later).

In Delphi, we can declare an interface like so:

```delphi
type
  ICalculator = interface (IInvokable)
    ['{9A6B8BED-CEB2-4E09-87D4-4A16F496E5FE}']
    /// add two signed 32-bit integers
    function Add(n1, n2: integer): integer;
  end;
```

It just sounds like a class definition, but, as you can see:

- It is named ICalculator, and not TCalculator: it is a common convention to start an interface name with a I, to make a difference with a T for a class or other implementation-level type definition;
- There is no visibility attribute (no private / protected / public / published keywords): in fact, it is just as if all methods were published;
- There is no fields, just methods (fields are part of the implementation, not of the interface): in fact, you can have properties in your interface definition, but those properties shall redirect to existing
getter and setter methods, via read and write keywords;  
- There is a strange number below the interface name, called a GUID: this is an unique identifier of the interface - you can create such a genuine constant on the editor cursor position by pressing Ctrl + Shift + G in the Delphi IDE;  
- But the methods are just defined as usual.

15.1.2. Implementing an interface with a class

Now that we have an interface, we need to create an implementation.

Our interface is very basic, so we may implement it like this:

```delphi
type TServiceCalculator = class(TInterfacedObject, ICalculator)
protected
  fBulk: string;
public
  function Add(n1, n2: integer): integer;
  procedure SetBulk(const aValue: string);
end;

function TServiceCalculator.Add(n1, n2: integer): integer;
begin
  result := n1+n2;
end;

procedure TServiceCalculator.SetBulk(const aValue: string);
begin
  fBulk := aValue;
end;
```

You can note the following:

- We added ICalculator name to the class() definition: this class inherits from TInterfacedObject, and implements the ICalculator interface;  
- Here we have protected and public keywords - but the Add method can have any visibility, from the interface point of view: it will be used as implementation of an interface, even if the method is declared as private in the implementation class;  
- There is a SetBulk method which is not part of the ICalculator definition - so we can add other methods to the implementation class, and we can even implement several interfaces within the same method (just add other interface names after like class(TInterfacedObject, ICalculator, IAnotherInterface));  
- There a fBulk protected field member within this class definition, which is not used either, but could be used for the class implementation.  
- Here we have to code an implementation for the TServiceCalculator.Add() method (otherwise the compiler will complain for a missing method), whereas there is no implementation expected for the ICalculator.Add method - it is perfectly "abstract".

15.1.3. Using an interface

Now we have two ways of using our TServiceCalculator class:

- The classic way;  
- The abstract way (using an interface).

The "classic" way, using an explicit class instance:

```delphi
function MyAdd(a, b: integer): integer;
var Calculator: TServiceCalculator;
```
begin
  Calculator := TServiceCalculator.Create;
  try
    result := Calculator.Add(a,b);
  finally
    Calculator.Free;
  end;
end;

Note that we used a try..finally block to protect the instance memory resource.

Then we can use an interface:

function MyAdd(a,b: integer): integer;
var
calculator: ICalculator;
begin
  calculator := TServiceCalculator.Create;
  result := calculator.Add(a,b);
end;

What's up over there?
- We defined the local variable as ICalculator: so it will be an interface, not a regular class instance;
- We assigned a TServiceCalculator instance to this interface variable: the variable will now handle the instance life time;
- We called the method just as usual - in fact, the computation is performed with the same exact expression: result := calculator.Add(a,b);
- We do not need any try...finally block here: in Delphi, interface variables are reference-counted: that is, the use of the interface is tracked by the compiler and the implementing instance, once created, is automatically freed when the compiler realizes that the number of references to a given interface variable is zero;
- And the performance cost is negligible: this is more or less the same as calling a virtual method (just one more redirection level).

In fact, the compiler creates an hidden try...finally block in the MyAdd function, and the instance will be released as soon as the Calculator variable is out of scope. The generated code could look like this:

function MyAdd(a,b: integer): integer;
var
calculator: TServiceCalculator;
begin
  calculator := TServiceCalculator.Create;
  try
    calculator.FRefCount := 1;
    result := calculator.Add(a,b);
  finally
    dec(calculator.FRefCount);
    if calculator.FRefCount=0 then
      calculator.Free;
  end;
end;

Of course, this is a bit more optimized than this (and thread-safe), but you have got the idea.

15.1.4. There is more than one way to do it
One benefit of interfaces we have already told about, is that it is "orthogonal" to the implementation.

In fact, we can create another implementation class, and use the same interface:

type
  TOtherServiceCalculator = class(TInterfacedObject, ICalculator)

Here the computation is not the same: we use \( n2+n1 \) instead of \( n1+n2 \)... of course, this will result into the same value, but we can use this another method for our very same interface, by using its `TOtherServiceCalculator` class name:

```pascal
function MyOtherAdd(a,b: integer): integer;
var
  Calculator: ICalculator;
begin
  Calculator := TOtherServiceCalculator.Create;
  result := Calculator.Add(a,b);
end;
```

15.1.5. Here comes the magic

Now you may begin to see the point of using interfaces in a client-server framework like ours.

Our mORMot is able to use the same interface definition on both client and server side, calling all expected methods on both sides, but having all the implementation logic on the server side. The client application will transmit method calls (using JSON instead of much more complicated XML/SOAP) to the server (using a "fake" implementation class created on the fly by the framework), then the execution will take place on the server (with obvious benefits), and the result will be sent back to the client, as JSON. The same interface can be used on the server side, and in this case, execution will be in-place, so very fast.

By creating a whole bunch of interfaces for implementing the business logic of your project, you will benefit of an open and powerful implementation pattern.

More on this later on... first we'll take a look at good principles of playing with interfaces.
15.2. SOLID design principles

The acronym SOLID is derived from the following OOP principles (quoted from the corresponding Wikipedia article):

- **Single responsibility principle**: the notion that an object should have only a single responsibility;
- **Open/closed principle**: the notion that "software entities ... should be open for extension, but closed for modification";
- **Liskov substitution principle**: the notion that "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program" - also named as "design by contract";
- **Interface segregation principle**: the notion that "many client specific interfaces are better than one general purpose interface.";
- **Dependency inversion principle**: the notion that one should "Depend upon Abstractions. Do not depend upon concretions.". **Dependency injection** is one method of following this principle, which is also called **Inversion Of Control** (aka IoC).

If you have some programming skills, those principles are general statements you may already found out by yourself. If you start doing serious object-oriented coding, those principles are best-practice guidelines you will definitively gain following.

They certainly help to fight the three main code weaknesses:

- **Rigidity**: Hard to change something because every change affects too many other parts of the system;
- **Fragility**: When you make a change, unexpected parts of the system break;
- **Immobility**: Hard to reuse in another application because it cannot be disentangled from the current application.

15.2.1. Single Responsibility Principle

When you define a class, it shall be designed to implement only one feature. The so-called feature can be seen as an "axis of change" or a "a reason for change".

Therefore:

- One class shall have only one reason that justifies changing its implementation;
- Classes shall have few dependencies on other classes;
- Classes shall be abstract from the particular layer they are running - see **Multi-tier architecture** (page 88).

For instance, a TRectangle object should not have both ComputeArea and Draw methods defined at once - they will define two responsibilities or axis of change: the first responsibility is to provide a mathematical model of a rectangle, and the second is to render it on GUI.

15.2.1.1. Splitting classes

To take an example from real coding, imagine you define a communication component. You want to communicate, say, with a bar-code scanner peripheral. You may define a single class, e.g. TBarcodeScanner, supporting such device connected over a serial port. Later on, the manufacturer deprecates the serial port support, since no computer still have it, and offer only USB models in its catalog. You may inherit from TBarcodeScanner, and add USB support.
SOLID Principles - Single Responsibility: Single-to-rule-them-all class

But in practice, this new TUsbBarCodeScanner class is difficult to maintain, since it will inherit from serial-related communication. So you start splitting the class hierarchy, using an abstract parent class:

```
TAbstractBarcodeScanner = class(TComponent)
protected
  function ReadChar: byte; virtual; abstract;
  function ReadFrame: TProtocolFrame; virtual; abstract;
  procedure WriteChar(aChar: byte); virtual; abstract;
  procedure WriteFrame(const aFrame: TProtocolFrame); virtual; abstract;
...
```

Then, TSerialBarCodeScanner and TUsbBarCodeScanner classes will override those classes, according to the final implementation.

In fact, this approach is cleaner. But it is not perfect either, since it may be hard to maintain and extend. Imagine the manufacturer is using a standard protocol for communication, whatever USB or Serial connection is used. You will put this communication protocol (e.g. its state machine, its stream computation, its delaying settings) in the TAbstractBarcodeScanner class. But perhaps they will be diverse flavors, in TSerialBarCodeScanner or TUsbBarCodeScanner, or even due to diverse models and features (e.g. if it supports 2D or 3D bar-codes).

It appears that putting everything in a single class is not a good idea. Splitting protocol and communication appears to be preferred. Each "axis of change" - i.e. every aspect which may need modifications - requires its own class. Then the T*BarcodeScanner classes will compose protocols and communication classes within a single component.

Imagine we have two identified protocols (named BCP1 and BCP2), and two means of communication (serial and USB). So we will define the following classes:
**SOLID Principles - Single Responsibility: Spliting protocol and communication**

Then, we may define our final classes and components as such:

```pascal
type
  TAbstractBarcodeConnection = class
    protected
      function ReadChar: byte; override;
      procedure WriteChar(aChar: byte); override;
    ...
  end;

  TAbstractBarcodeProtocol = class
    protected
      fConnection: TAbstractBarcodeConnection;
      function ReadFrame: TProtocolFrame; override;
      procedure WriteFrame(const aFrame: TProtocolFrame); override;
    ...
  end;

  TAbstractBarcodeScanner = class(TComponent)
    protected
      fProtocol: TAbstractBarcodeProtocol;
      fConnection: AbstractBarcodeConnection;
    ...
  end;
```

And each actual inherited class will initialize the protocol and connection according to the expected model:

```pascal
constructor TSerialBarcodeScanner.Create(const aComPort: string; aBitRate: integer);
begin
  fConnection := TSerialBarcodeConnection(aComPort,aBitRate);
  fProtocol := TBCP1BarcodeProtocol.Create(fConnection);
end;
```

Here, we inject the connection instance to the protocol, since the later may need to read or write some bytes on the wire, when needed.

Another example is how our database classes are defined in SynDB.pas - see *External SQL database access* (page 239):

- The **connection properties** feature is handled by TSQLDBConnectionProperties classes;
- The actual **living connection** feature is handled by TSQLDBConnection classes;
- And **database requests** feature is handled by TSQLDBStatement instances using dedicated NewConnection / ThreadSafeConnection / NewStatement methods.

Therefore, you may change how a database connection is defined (e.g. add a property to a TSQLDBConnectionProperties child), and you won't have to change the statement implementation itself.

**15.2.1.2. Do not mix UI and logic**

Another practical "Single Responsibility Principle" smell may appear in your uses clause.

If your data-only or peripheral-only unit starts like this:

```pascal
unit MyDataModel;
uses
  Winapi.Windows,
  mORMot,
  ...
```

It will induce a dependency about the Windows Operating System, whereas your data will certainly benefit from being OS-agnostic. Our todays compiler (Delphi or FPC) targets several OS, so coupling our data to the actual Windows unit does show a bad design.

Similarly, you may add a dependency to the VCL, via a reference to the Forms unit.
If your data-only or peripheral-only unit starts like the following, beware!

```pascal
unit MyDataModel;

uses
  Winapi.Messages,
  Vcl.Forms,
  mORMot,
  ...
```

If you later want to use FMX, or LCL (from Lazarus) in your application, or want to use your MyDataModel unit on a pure server application without any GUI, hosted on Windows - or even better on Linux/BSD - you are stuck.

Note that if you are used to developed in RAD mode, the units generated by the IDE wizards come with some default references in the uses clause of the generated .pas file. So take care of not introducing any coupling to your own business code!

As a general rule, our ORM/SOA framework source code tries to avoid such dependencies. All OS-specificities are centralized in our SynCommons.pas unit, and there is no dependency to the VCL when it is not mandatory, e.g. in mORMot.pas.

Following the RAD approach, you may start from your UI, i.e. defining the needed classes in the unit where you visual form (may be VCL or FMX) is defined. Don't follow this tempting, but dangerous path!

Code like the following may be accepted for a small example (e.g. the one supplied in the SQlite3\Samples sub-folder of our repository source code tree), but is to be absolutely avoided for any production ready mORMot-based application:

```pascal
interface

uses
  Vcl.Controls, Vcl.Forms, Vcl.Dialogs, mORMot, mORMotSQLite3;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    fModel: TSQModel;
    fDatabase: TSQLRestServerDB;
  public
  end;

implementation

procedure TForm1.FormCreate(Sender: TObject);
begin
  fModel := TSQModel.Create([TSQMyOwnRecord], 'root');
  fDatabase := TSQLRestServerDB.Create(fModel, ChangeFileExt(paramstr(0), '.db'));
end;
```

In your actual project units, when you define an ORM or SOA class, never include GUI methods within. In fact, the fact that our TSQLRecord class definitions are common to both Client and Server sides makes this principle mandatory. You should not have any GUI related method on the Server side, and the Client side could use the objects instances with several GUI implementations (Delphi Client, AJAX Client...).

Therefore, if you want to change the GUI, you won't have to recompile the TSQLRecord class and the
associated database model. If you want to deploy your server on a Linux box (using e.g. CrossKylix or FPC as compiler), you could reuse your very same code, since you do not have reference to the VCL in your business code.

This *single responsibility principle* may sound simple and easy to follow (even obvious), but in fact, it is one of the hardest principles to get right. Naturally, we tend to join responsibilities in our class definitions. Our framework architecture will enforce you, by its Client-Server nature and all its high-level methods involving interface, to follow this principle, but it is always up to the end coder to design properly his/her types.

### 15.2.2. Open/Closed Principle

When you define a class or a unit, at the same time:

- They shall be *open for extension*;
- But *closed for modification*.

It means that you may be able to extend your existing code, without breaking its initial behavior. Some other guidelines may be added, but you got the main idea.

Conformance to this open/closed principle is what yields the greatest benefit of OOP, i.e.:

- Code re-usability;
- Code maintainability;
- Code extendibility.

Following this principle will make your code far away from a regular RAD style. But benefits will be huge.

#### 15.2.2.1. Applied to our framework units

When designing our ORM/ SOA set of units, we tried to follow this principle. In fact, you should not have to modify its implementation. You should define your own units and classes, without the need to hack the framework source code.

Even if *Open Source* paradigm allows you to modify the supplied code, this shall not be done unless you are either fixing a bug or adding a new common feature. This is in fact the purpose of our https://synopse.info..web site, and most of the framework enhancements have come from user requests.

The framework Open Source license - see below (page 643) - may encourage user contributions in order to fulfill the Open/closed design principle:

- Your application code extends the *Synopse mORMot Framework* by defining your own classes or event handlers - this is how it is *open for extension*;
- The main framework units shall remain inviolate, and common to all users - this illustrates the *closed for modification* design.

As a beneficial side effect, this principle will ensure that your code will be ready to follow the framework updates (which are quite regular). When a new version of *mORMot* is available, you should be able to retrieve it for free from our web site, replace your files locally, then build a new enhanced version of your application, with the benefit of all included fixes and optimizations. Even the source code repository is available - at https://synopse.info/fossil.. or from https://github.com/synopse/mORMot.. - and allows you to follow the current step of development of the framework.
In short, abstraction is the key to peace of mind. All your code shall not depend on a particular implementation.

15.2.2.2. Open/Closed in practice

In order to implement this principle, several conventions could be envisaged:
- You shall better define some abstract classes, then use specific overridden classes for each and every implementation: this is for instance how Client-Server classes were implemented - see Client-Server process (page 318);
- All object members shall be declared private or protected - this is a good idea to use Service-Oriented Architecture (SOA) (page 90) for defining server-side process, and/or make the TSQLRecord published properties read-only and using some client-side constructor with parameters;
- No singleton nor global variable - ever;
- RTTI is dangerous - that is, let our framework use RTTI functions for its own cooking, but do not use it in your code.

In our previous bar-code scanner class hierarchy, we will therefore define the

type
begin
  TAbstractBarcodeScanner = class(TComponent)
  protected
    fProtocol: TAbstractBarcodeProtocol;
    fConnection: AbstractBarcodeConnection;
    ...;
  public
    property Connection: AbstractBarcodeConnection read fConnection;
    ...;
end;

In this code, the actual variables are stored as protected fields, with only getters (i.e. read) in the public section. There is no setter (i.e. write) attribute, which may allow to change the fProtocol/fConnection instances in user code. You can still access those fields (it is mandatory in your inherited constructors), but user code should not use it.

As stated above - see SOLID Principles - Single Responsibility: Spliting protocol and communication (page 390) - having dedicated classes for defining protocol and connection will also help implementing the open/closed principle. You will be able to define a new class, combining its own protocol and connection class instances, so it will be Open for extension. But you will not change the behavior of a class, by inheriting it: since protocol and connection are uncoupled, and used via composition in a dedicated class, it will be Closed for modification.

Using the newest sealed directive for a class may ensure that your class definition will follow this principle. If the class method or property is sealed, you will not be able to change its behavior in its inherited types, even if you are tempted to.

15.2.2.3. No Singleton nor global variables

About the singleton pattern, you should better always avoid it in your code. In fact, a singleton was a C++ (and Java) hack invented to implement some kind of global variables, hidden behind a static class definition. They were historically introduced to support mixed mode of application-wide initialization (mainly allocate the stdio objects needed to manage the console), and were abused in business logic.

Once you use a singleton, or a global variable, you will miss most of the benefit of OOP. A typical use of singleton is to register some class instances globally for the application. You may see some framework - or some part of the RTL - which will allow such global registration. But it will eventually
void most benefits of proper dependency injection - see below (page 400) - since you will not be able to have diverse resolution of the same class.

For instance, if your database properties, or your application configuration are stored within a singleton, or a global variable, you will certainly not be able to use several database at once, or convert your single-user application with its GUI into a modern multi-user AJAX application:

```pascal
var
    DBServer: string = 'localhost';
    DBPort: integer = 1426;
    UITextColor: TColor = clNavy;
    UITextSize: integer = 12;
```

Such global variables are a smell of a broken Open/Closed Principle, since your project will definitively won’t be open for extension. Using a static class variable (as allowed in newer version of Delphi), is just another way of defining a global variable, just adding the named scope of the class type.

Even if you do not define some global variable in your code, you may couple your code from an existing global variable. For instance, defining some variables with your `TMainForm = class(TForm)` class defined in the IDE, then using its global `MainForm: TMainForm` variable, or the `Application.MainForm` property, in your code. You will start to feel not right, when the unit where your `TMainForm` is defined will start to appear in your business code uses clause... just another global variable in disguise!

In our framework, we tried to never use global registration, but for the cases where it has been found safe to be implemented, e.g. when RTTI is cached, or JSON serialization is customized for a given type. All those informations will be orthogonal to the proper classes using them, so you may find some global variables in the framework units, only when it is worth it. For instance, we split `TSQLRecord`'s information into a `TSQLRecordProperties` for the shared intangible RTTI values, and `TSQLModelRecordProperties` instances, one per `TSQLModel`, for all the `TSQLModel/TSQLRest` specific settings - see Several Models (page 171).

### 15.2.3. Liskov Substitution Principle

Even if her name is barely unmemorable, Barbara Liskov is a great computer scientist, we should better learn from. It is worth taking a look at her presentation at [https://www.youtube.com/watch?v=GDVAHA0oyJU.](https://www.youtube.com/watch?v=GDVAHA0oyJU.)

The "Liskov substitution principle" states that, if TChild is a subtype of TParent, then objects of type TParent may be replaced with objects of type TChild (i.e., objects of type TChild may be substitutes for objects of type TParent) without altering any of the desirable properties of that program (correctness, task performed, etc.).

The example given by Barbara Liskov was about stacks and queues: even if both do share Push and Pop methods, they should not inherit from a single parent type, since the storage behavior of a stack is quite the contrary of a queue. In your program, if you start to replace a stack by a queue, you will meet strange behaviors, for sure. According to proper top-bottom design flow, both types should be uncoupled. You may implement a `TFastStack` class using an in-memory list for storage, or another `TPersistedStack` class using a remote SQL engine, but both will have to behave like a `TStack`, i.e. according to the last-in first-out (LIFO) principle. On the other hand, any class implementing a `queue` type should follow the the first-in first-out (FIFO) order, whatever kind of storage is used.

In practical Delphi code, relying on abstractions may be implemented by two means:
- Using only abstract parent class variables when consuming objects;
- Using interface variable instead of class implementations.

Here, we do not use inheritance for sharing implementation code, but for defining an expected behavior. Sometimes, you may break the Liskov Substitution principle in implementation methods which will be coded just to gather some reusable pieces of code (the inheritance for implementation pattern), preparing some behavior which may be used only by some of the subtypes. Such "internal" virtual methods of a subtype may change the behavior of its inherited method, for the sake of efficiency and maintainability. But with this kind of implementation inheritance, which is closer to plumbing than designing, methods should be declared as protected, and not published as part of the type definition.

By the way, this is exactly what interface type definitions have to offer. You can inherit from another interface, and this kind of polymorphism should strictly follow the Liskov Substitution principle. Whereas the class types, implementing the interfaces, may use some protected methods which may break the principle, for the sake of code efficiency.

In order to fulfill this principle, you should:

- Properly name (and comment) your class or interface definition: having Push and Pop methods may be not enough to define a contract, so in this case type inheritance will define the expected expectation - as a consequence, you should better stay away from "duck typing" patterns, and dynamic languages, but rely on strong typing;
- Use the "behavior" design pattern, when defining your objects hierarchy - for instance, if a square may be a rectangle, a TSquare object is definitively not a TRectangle object, since the behavior of a TSquare object is not consistent with the behavior of a TRectangle object (square width always equals its height, whereas it is not the case for most rectangles);
- Write your tests using abstract local variables (and this will allow test code reuse for all children classes);
- Follow the concept of Design by Contract, i.e. the Meyer's rule defined as "when redefining a routine [in a derivative], you may only replace its precondition by a weaker one, and its postcondition by a stronger one" - use of preconditions and postconditions also enforce testing model;
- Separate your classes hierarchy: typically, you may consider using separated object types for implementing persistence and object creation (this is the common separation between Factory and Repository patterns).

15.2.3.1. Use parent classes

Within our framework, it will signify that TSQLRestServer or TSQLRestClient instances can be substituted to a TSQLRest object. Most ORM methods, especially at TSQLRecord level, expect an abstract TSQLRest parameter to be supplied - see Working with Objects (page 144).

For instance, you may write:

```pascal
var anyRest: TSQLRest;
    ID: TID;
    rec1, rec2: TSQLMyRecord;

ID := anyRest.Add(rec1, true);
rec2 := TSQLMyRecord.Create(anyRest, ID);
```

And you may set any kind of actual class instance to anyRest, either a local stored database engine, or a HTTP remote access:

```pascal
anyRest := TSQLRestServerDB.Create(aModel, 'mydatabase.db ');
anyRest := TSQLHttpClient.Create('1.2.3.4', '8888', aModel, false);
```
You may even find in the dddInfraSettings.pas unit a powerful TRestSettings.NewRestInstance() method which is able to instantiate the needed TSQLRest inherited class from a set of JSON settings, i.e. either a TSQLHttpClient, or a local TSQLRestServerFullMemory, or a TSQLRestServerDB - the later either with a local SQLite3 database, an external SQL engine, or an external NoSQL/MongoDB database.

Your code shall refer to abstractions, not to implementations. By using only methods and properties available at classes parent level, your code won't need to change because of a specific implementation.

15.2.3.2. I'm your father, Luke

You should note that, in the Liskov substitution principle definition, "parent" and "child" are no absolute. Which actual class is considered as "parent" may depend on the context use.

Most of the time, the parent may be the highest class in the hierarchy. For instance, in the context of a GUI application, you may use the most abstract class to access the application data, may it be stored locally, or remotely accessed over HTTP.

But when you initialize the class instance of a local stored server, you may need to setup the actual data storage, e.g. the file name or the remote SQL/NoSQL settings. In this context, you will need to access the "child" properties, regardless of the "parent" abstract use which will take care later on in the GUI part of the application.

Furthermore, in the context of data replication, server side or client side will have diverse behavior. In fact, they may be used as master or slave database, so in this case, you may explicitly define server or client class in your code. This is what our ORM does for its master/slave replication - see Master/slave replication (page 180).

If we come back to our bar-code scanner sample, most of your GUI code may rely on TAbstractBarcodeScanner components. But in the context of the application options, you may define the internal properties of each "child" class - e.g. the serial or USB port name, so in this case, your new "parent" class may be either TSerialBarCodeScanner or TUsbCodeScanner, or even better the TSerialBarcodeConnection or TUsbBarcodeConnection properties, to fulfill Single Responsibility principle.

15.2.3.3. Don't check the type at runtime

Some patterns shall never appear in your code. Otherwise, code refactoring should be done as soon as possible, to let your project be maintainable in the future.

Statements like the following are to be avoided, in either the parents' or the childs' methods:

```pascal
procedure TAbstractBarcodeScanner.SomeMethod;
begin
  if self is TSerialBarcodeScanner then
    begin
      ....
    end
  else
    if self is TUsbBarcodeScanner then
      ...
```

Or, in its disguised variation, using an enumerated item:

```pascal
case fProtocol.MeanOfCommunication of
  meanSerial: begin
  ....
```
This later piece of code does not check self, but the fProtocol protected field. So even if you try to implement the Single Responsibility principle, you may still be able to break Liskov Substitution!

Note that both patterns will eventually break the Single Responsibility principle: each behavior shall be defined in its own child class methods. As the Open/Closed principle will also be broken, since the class won't be open for extension, without touching the parent class, and modify the nested if self is T* then ... or case fProtocol.* of ... expressions.

15.2.3.4. Partially abstract classes

Another code smell may appear when you define a method which will stay abstract for some children, instantiated in the project. It will imply that some of the parent class behavior is not implemented at this particular hierarchy level. So you will not be able to use all the parent's methods, as will be expected by the Liskov Substitution principle.

Note that the compiler will complain for it, hinting that you are creating a class with abstract methods. Never ignore such hints - which may benefit for being handled as errors at compilation time. The (in)famous "Abstract Error" error dialog, which may appear at runtime, will reflect this bad code implementation. When it occurs on a server application without GUI... you got a picture of the terror, I guess...

A more subtle violation of Liskov may appear if you break the expectation of the parent class. The following code, which emulates a bar-code reader peripheral by sending the frame by email for debugging purpose (why not?), clearly fails the Design by Contract approach:

```
TEMai1EmulatedBarcodeProtocol = class(TAbstractBarcodeProtocol)
  protected
  function ReadFrame: TProtocolFrame; override;
    procedure WriteFrame(const aFrame: TProtocolFrame); override;
    ...

  function TEMai1EmulatedBarcodeProtocol.ReadFrame: TProtocolFrame;
  begin
    raise EBarcodeException.CreateUTF8('%ReadFrame is not implemented!',[self]);
  end;

  procedure TEMai1EmulatedBarcodeProtocol.WriteFrame(const aFrame: TProtocolFrame);
  begin
    SendEmail(fEmailNotificationAddress,aFrame.AsString);
  end;

We expected this class to fully implement the TAbstractBarcodeProtocol contract, whereas calling TEMai1EmulatedBarcodeProtocol.ReadFrame will not be able to read any data frame, but will raise an exception. So we can not use this TEMai1EmulatedBarcodeProtocol class as replacement to any other TAbstractBarcodeProtocol class, otherwise it will fail at runtime.

A correct implementation may perhaps to define a TFakeBarcodeProtocol class, implementing all the parent methods via a set of events or some text-based scenario, so that it will behave just like a correct TAbstractBarcodeProtocol class, in the full extend of its expectations.

15.2.3.5. Messing units dependencies

Last but not least, if you need to explicitly add child classes units to the parent class unit uses clause, it looks like if you just broke the Liskov Substitution principle.

```
unit AbstractBarcodeScanner;
```
If your code is like this, you will have to remove the reference to the inherited classes, for sure.

Even a dependency to one of the low-level implementation detail is to be avoided:

```pascal
unit AbstractBarcodeScanner;

uses
  Windows,
  SysUtils,
  Classes,
  SerialBarcodeScanner; // Barbara complains: “it smells”!
  UsbBarcodeScanner; // Barbara complains: “it smells”!
...
```

Your abstract parent class should not be coupled to a particular Operating System, or a mean of communication, which may not be needed. Why will you add a dependency to raw RS-232 communication protocol, which is very likely to be deprecated?

One way of getting rid of this dependency is to define some abstract types (e.g. enumerations or simple structures like record), which will then be translated into the final types as expected by the ComPort.pas or Windows.pas units. Consider putting all the child classes dependencies at constructor level, and/or use class composition via the Single Responsibility principle so that the parent class definition will not be polluted by implementation details of its children.

You may also use a registration list, maintained by the parent unit, which may be able to register the classes implementing a particular behavior at runtime. Thanks to Liskov, you will be able to substitute any parent class by any of its inherited implementation, so defining the types at runtime only should not be an issue.

### 15.2.3.6. Practical advantages

The main advantages of this coding pattern are the following:

- Thanks to this principle, you will be able to stub or mock an interface or a class - see below (page 406) - e.g. uncouple your object persistence to the actual database it runs on: this principle is therefore mandatory for implementing unitary testing to your project;
- Furthermore, testing will be available not only at isolation level (testing each child class), but also at abstracted level, i.e. from the client point of view - you can have implementation which behave correctly when tested individually, but which failed when tested at higher level if the Liskov principle was broken;
- As we have seen, if this principle is violated, the other principles are very likely to be also broken - e.g. the parent class will need to be modified whenever a new derivative of the base class is defined (violation of the Open/Closed principle), or your class types may implement more than one behavior at a time (violation of the Single Responsibility principle);
- Code re-usability is enhanced by method re-usability: a method defined at a parent level does not require to be implemented for each child.

The SOA and ORM concepts, as implemented by our framework, try to be compliant with the Liskov substitution principle. It is true at class level for the ORM, but a more direct Design by Contract implementation pattern is also available, since the whole SOA stack involves a wider usage of interfaces in your projects.
### 15.2.4. Interface Segregation Principle

This principle states that once an interface has become too 'fat' it shall be split into smaller and more specific interfaces so that any clients of the interface will only know about the methods that pertain to them. In a nutshell, no client should be forced to depend on methods it does not use.

As a result, it will help a system stay decoupled and thus easier to re-factor, change, and redeploy.

#### 15.2.4.1. Consequence of the other principles

*Interface segregation* should first appear at class level. Following the *Single Responsibility* principle, you are very likely to define several smaller classes, with a small extent of methods. Then use dedicated types of class, relying on composition to expose its own higher level set of methods.

The bar-code class hierarchy illustrates this concept. Each `T*BarcodeProtocol` and `T*BarcodeConnection` class will have its own set of methods, dedicated either to protocol handling, or data transmission. Then the `T*BarCodeScanner` classes will *compose* those smaller classes into a new class, with a single event handler:

```delphi
type
   TOnBarcodeScanned = procedure(Sender: TAbstractBarcodeScanner; const Barcode: string) of object;
   TAbstractBarcodeScanner = class(TComponent)
      property OnBarcodeScanned: TOnBarcodeScanned read fOnBarcodeScanned write fOnBarcodeScanned;
   ...
```

This single `OnBarcodeScanned` event will be the published property of the component. Both protocol and connection details will be hidden within the internal classes. The final application will use this event, and react as expected, without actually knowing anything about the implementation details.

#### 15.2.4.2. Using interfaces

The SOA part of the framework allows direct use of interface types to implement services. This great Client-Server SOA implementation pattern - see *Server side Services* (page 372) - helps decoupling all services to individual small methods. In this case also, the stateless used design will also reduce the use of 'fat' session-related processes: an object life time can be safely driven by the *interface* scope.

By defining *Delphi interface* instead of plain class, it helps creating small and business-specific contracts, which can be executed on both client and server side, with the same exact code.

Since the framework makes interface consumption and publication very easy, you won't be afraid of exposing your implementation classes as small pertinent interface.

For instance, if you want to publish a third-party API, you may consider publishing dedicated interfaces, each depending on every API consumer expectations. So your main implementation logic won't be polluted by how the API is consumed, and, as correlative, the published API may be closer to each particular client needs, without been polluted by the other client needs. DDD will definitively benefit for *Interface Segregation*, since this principle is the golden path to avoid *domain leaking* - see below (page 600).

### 15.2.5. Dependency Inversion Principle

Another form of decoupling is to invert the dependency between high and low level of a software design:

- High-level modules should not depend on low-level modules. Both should depend on abstractions;
- Abstractions should not depend upon details. Details should depend upon abstractions.
The goal of the dependency inversion principle is to decouple high-level components from low-level components such that reuse with different low-level component implementations becomes possible. A simple implementation pattern could be to use only interfaces owned by, and existing only with the high-level component package.

This principle results in Inversion Of Control (aka IoC): since you rely on the abstractions, and try not to depend upon concretions (i.e. on implementation details), you should first concern by defining your interfaces.

15.2.5.1. Upside Down Development

In conventional application architecture, lower-level components are designed to be consumed by higher-level components which enable increasingly complex systems to be built. This design limits the reuse opportunities of the higher-level components, and certainly breaks the Liskov substitution principle.

For our bar-code reader sample, we may be tempted to start from the final TSerialBarcodeScanner we need in our application. We were asked by our project leader to allow bar-code scanning in our flagship application, and the extend of the development has been reduced to support a single model of device, in RS-232 mode - this may be the device already owned by our end customer.

This particular customer may have found some RS-232 bar-code relics from the 90s in its closets, but, as an experience programmer, you know that the next step will be to support USB, in a very close future. All this bar-code reading stuff will be marketized by your company, so it is very likely that another customer will very soon ask for using its own brand new bar-code scanners... which will support only USB.

So you will modelize your classes as with SOLID Principles - Single Responsibility: Abstract parent class (page 390) and SOLID Principles - Single Responsibility: Spliting protocol and communication (page 390). Even if the TUsbBarcodeScanner - and its correlative TUsbBarCodeConnection class - is not written, nor tested (you do not even have an actual USB bar-code scanner to do proper testing yet!), you are prepared for it.

When you will eventually add USB support, the UI part of the application won't have to be touched. Just implementing your new inherited class, leveraging all previous coding. Following Dependency Inversion from the beginning will definitively save your time. Even in an Agile kind of process - where "Responding to change" is most valuable - the small amount of work on implementing first from the abstraction with the initial implementation will be very beneficial.

In fact, this Dependency Inversion principle is a prerequisite for proper Test-Driven Design. Following this TDD pattern, you first write your test, then fail your test, then write the implementation. In order to write the test, you need the abstracted interface of the feature to be available. So you will start from the abstraction, then write the concretion.

15.2.5.2. Injection patterns

In other languages (like Java or .Net), various patterns such as Plug-in, Service Locator, or Dependency Injection are then employed to facilitate the run-time provisioning of the chosen low-level component implementation to the high-level component.

Our Client-Server architecture facilitates this decoupling pattern for its ORM part, and allows the use of native Delphi interFace to call services from an abstract factory, for its SOA part.

A set of dedicated classes, defined in mORMot.pas, allows to leverage IoC: see e.g.
TInjectableObject, TInterfaceResolver, TInterfaceResolverForSingleInterface and TInterfaceResolverInjected, which may be used in conjunction with TInterfaceStub or TServiceContainer high-level mocking and SOA features of the framework - see below (page 406) and below (page 419).

15.3. Circular reference and (zeroing) weak pointers

15.3.1. Weak pointers

The memory allocation model of the Delphi interface type uses some kind of Automatic Reference Counting (ARC). In order to avoid memory and resource leaks and potential random errors in the applications (aka the terrible EAccessViolation exception on customer side) when using Interfaces (page 385), a SOA framework like mORMot has to offer so-called Weak pointers and Zeroing Weak pointers features.

By default in Delphi, all references are defined:
- as weak references for pointer and class instances;
- with explicit copy for low-level value types like integer, Int64, currency, double or record (and old deprecated object or shortstring);
- via copy-on-write with reference counting for high-level value types (e.g. string, widestring, variant or a dynamic array - with the exception of tuned memory handling for TDocVariant custom variant type (page 112));
- as strong reference with reference counting for interface instances.

The main issue with strong reference counting is the potential circular reference problem. This occurs when an interface has a strong pointer to another, but the target interface has a strong pointer back to the original. Even when all other references are removed, they still will hold on to one another and won’t be released. This can also happen indirectly, by a chain of objects that might have the last one in the chain referring back to an earlier object.

See the following interface definition for instance:

```delphi
interface IParent
procedure SetChild(const Value: IChild);
function GetChild: IChild;
function HasChild: boolean;
property Child: IChild read GetChild write SetChild;
end;

interface IChild
procedure SetParent(const Value: IParent);
function GetParent: IParent;
property Parent: IParent read GetParent write SetParent;
end;
```

The following implementation will definitively leak memory:

```delphi
procedure TParent.SetChild(const Value: IChild);
begin
  FChild := Value;
end;

procedure TChild.SetParent(const Value: IParent);
begin
  FParent := Value;
end;
```

In Delphi, most common kind of reference-copy variables (i.e. variant, dynamic array or string)
solve this issue by implementing *copy-on-write*. Unfortunately, this pattern is not applicable to *interface*, which are not *value* objects, but *reference* objects, tied to an implementation class, which can’t be copied.

One common solution is to use *Weak pointers*, by which the *interface* is assigned to a property without incrementing the reference count.

Note that garbage collector based languages (like Java or C#) do not suffer from this problem, since the circular references are handled by their memory model: objects lifetime are maintained globally by the memory manager. Of course, it will increase memory use, slowdown the process due to additional actions during allocation and assignments (all objects and their references have to be maintained in internal lists), and may slow down the application when garbage collector enters in action. In order to avoid such issues when performance matters, experts tend to pre-allocate and re-use objects: this is one common limitation of this memory model, and why *Delphi* is still a good candidate (like unmanaged C or C++ - and also *Objective C*) when it deals with performance and stability. In some cases (e.g. when using an object cache), such languages have to introduce some kind of "weak pointers", to allow some referenced objects to be reclaimed by garbage collection: but it is a diverse mechanism, under the same naming.

### 15.3.2. Handling weak pointers

In order to easily create a weak pointer, the following function was added to *mORMot.pas*:

```pascal
procedure SetWeak(aInterfaceField: PIInterface; const aValue: IInterface);
begin
  PPointer(aInterfaceField)^ := Pointer(aValue);
end;
```

It will assign the *interface* reference to a field by assigning the pointer of this instance to the internal field. It will by-pass the reference counting, so memory won't be leaked any more.

Therefore, it could be used as such:

```pascal
procedure TParent.SetChild(const Value: IChild);
begin
  SetWeak(@FChild,Value);
end;

procedure TChild.SetParent(const Value: IParent);
begin
  SetWeak(@FParent,Value);
end;
```

### 15.3.3. Zeroing weak pointers

But there are still some cases where it is not enough. Under normal circumstances, a *class* instance should not be deallocated if there are still outstanding references to it. But since weak references don't contribute to an *interface* reference count, a *class* instance can be released when there are outstanding weak references to it. Some memory leak or even random access violations could occur. A debugging nightmare...

In order to solve this issue, ARC's *Zeroing* Weak pointers come to mind. It means that weak references will be set to *nil* when the object they reference is released. When this happens, the automatic zeroing of the outstanding weak references prevents them from becoming dangling pointers. And *voilà*! No access violation any more!

Such a *Zeroing* ARC model has been implemented in *Objective C* by Apple, starting with Mac OS X 10.7.
Lion, in replacement (and/or addition) to the previous manual memory handling implementation pattern: in its Apple's flavor, ARC is available not only for interfaces, but for objects, and is certainly more sophisticated than the basic implementation available in the Delphi compiler: it is told (at least from the marketing paper point of view) to use some deep knowledge of the software architecture to provide an accurate access to all instances - whereas the Delphi compiler just relies on a out-of-scope pattern. In regard to classic garbage collector memory model, ARC is told to be much more efficient, due to its deterministic nature: Apple's experts ensure that it does make a difference, in term of memory use and program latency - which both are very sensitive on "modest" mobile devices. In short, thanks to ARC, your phone UI won't glitch during background garbage recycling. So mORMot will try to offer a similar feature, even if the Delphi compiler does not implement it (yet).

In order to easily create a so-called zeroing weak pointer, the following function was defined in mORMot.pas:

```pascal
procedure SetWeakZero(aObject: TObject; aObjectInterfaceField: PInterface; const aValue: IInterface);
```

A potential use case could be:

```pascal
procedure TParent.SetChild(const Value: IChild);
begin
  SetWeakZero(self,@FChild,Value);
end;
```

```pascal
procedure TChild.SetParent(const Value: IParent);
begin
  SetWeakZero(self,@FParent,Value);
end;
```

We also defined a class helper around the TObject class, to avoid the need of supplying the self parameter, but unfortunately, the class helper implementation is so buggy it won't be even able to compile before Delphi XE version of the compiler. But it will allow to write code as such:

```pascal
procedure TParent.SetChild(const Value: IChild);
begin
  SetWeak(@FChild,Value);
end;
```

For instance, the following code is supplied in the regression tests, and will ensure that weak pointers are effectively zeroed when SetWeakZero() is used:

```pascal
function TParent.HasChild: boolean;
begin
  result := FChild<>nil;
end;
```

```pascal
Child := nil; // here Child is destroyed
Check(Parent.HasChild=(aWeakRef=weakref),'ZEROed Weak');
```

Here, aWeakRef=weakref is true when SetWeak() has been called, and equals false when SetWeakZero() has been used to assign the Child element to its Parent interface.

**15.3.4. Weak pointers functions implementation details**

The SetWeak() function itself is very simple. The Delphi RTL/VCL itself use similar code when necessary.

But the SetWeakZero() function has a much more complex implementation, due to the fact that a list of all weak references has to be maintained per class instance, and set to nil when this referring instance is released.
The **mORMot** implementation tries to implement:

- Best performance possible when processing the *Zeroing* feature;
- No performance penalty for other classes not involved within weak references;
- Low memory use, and good scalability when references begin to define huge graphs;
- Thread safety - which is mandatory at least on the server side of our framework;
- Compatible with *Delphi* 6 and later (avoid syntax tricks like *generic*).

Some good existing implementations can be found on the Internet:

- **Andreas Hausladen** provided a classical and complete implementation at [http://andy.jgknet.de/blog/2009/06/weak-interface-references_](http://andy.jgknet.de/blog/2009/06/weak-interface-references_), using some nice tricks (like per-instance optional speed up using a void IWeakInterface interface whose VMT slot will refer to the references list), is thread-safe and is compatible with most *Delphi* versions - but it will slow down all TObject.FreeInstance calls (i.e. within Free / Destroy) and won't allow any overridden FreeInstance method implementation;
- **Vincent Parrett** proposed at [http://www.finalbuilder.com/Resources/Blogs/Postld/410/WeakReference-in-Delphi-solving-circular-interface.aspx](http://www.finalbuilder.com/Resources/Blogs/Postld/410/WeakReference-in-Delphi-solving-circular-interface.aspx), a generic-based solution (not thread-safe nor optimized for speed), but requiring to inherit from a base class for any class that can have a weak reference pointing to it;
- More recently, **Stefan Glienke** published at [http://delphisorcery.blogspot.fr/2012/06/weak-interface-references.html](http://delphisorcery.blogspot.fr/2012/06/weak-interface-references.html), another generic-based solution, not requiring to inherit from a base class, but not thread-safe and suffering from the same limitations related to TObject.FreeInstance.

The implementation included within **mORMot** uses several genuine patterns, when compared to existing solutions:

- It will hack the TObject.FreeInstance at the class VMT level, so will only slow down the exact class which is used as a weak reference, and not others (also its inherited classes won't be overridden) - and it will allow custom override of the virtual FreeInstance method;
- It makes use of our TDynArrayHashed wrapper to provide a very fast lookup of instances and references, without using generic definitions - hashing will start when it will be worth it, i.e. for any list storing more than 32 items;
- The unused vmtAutoTable VMT slot is used to handle the class-specific orientation of this feature (similar to TSQLRecordProperties lookup as implemented for *DI* # 2.1.3), for best speed and memory use.

See the TSetWeakZeroClass and TSetWeakZeroInstance implementation in mORMot.pas for the details.
15.4. Interfaces in practice: dependency injection, stubs and mocks

In order to fulfill the SOLID design principles (page 389), two features are to be available when handling interfaces:

- **Dependency injection or Inversion of Control** (aka IoC) - see Dependency Inversion Principle (page 400);
- **Stubbing and mocking** of interfaces for proper testing.

We will show now how mORMot provides all needed features for such patterns, testing a simple "forgot my password" scenario: a password shall be computed for a given user name, then transmitted via SMS, and its record shall be updated in the database.

15.4.1. Dependency Injection at constructors

A direct implementation of dependency injection at a class level can be implemented in Delphi as such:

- All external dependencies shall be defined as abstract interface;
- An external factory could be used to retrieve an interface instance, or class constructor shall receive the dependencies as parameters.

Using an external factory can be made within mORMot via TServiceFactory - see below (page 419). Automated dependency injection is also available via a set of classes, uncoupled from the SOA features of the framework, mainly TInjectableObject and TInterfaceResolver types, and their inherited classes - see below (page 417).

Here, we will use the more direct constructor-based pattern for a simple "forgot my password" scenario.

This is the class we want to test:

```delphi
TLoginController = class(TInterfacedObject, ILoginController)
protected
  fUserRepository: IUserRepository;
  fSmsSender: ISmsSender;
public
  constructor Create(const aUserRepository: IUserRepository;
                      const aSmsSender: ISmsSender);
  procedure ForgotMyPassword(const UserName: RawUTF8);
end;
```

The constructor will indeed inject its dependencies into its own instance:

```delphi
constructor TLoginController.Create(const aUserRepository: IUserRepository;
                                    const aSmsSender: ISmsSender);
begin
  fUserRepository := aUserRepository;
  fSmsSender := aSmsSender;
end;
```

The dependencies are defined with the following two interfaces (only the needed methods are listed here, but a real interface may have much more members, but not too much, to follow the interface segregation SOLID principle):

```delphi
IUserRepository = interface(IInvokable)
  ['{B21E5B21-28F4-4874-8446-800806DA07F}']
  function GetUserByName(const Name: RawUTF8): TUser;
  procedure Save(const User: TUser);
end;

ISmsSender = interface(IInvokable)
```
Note also that all those code will use a plain record as *Data Transfer Object* (DTO):

```pascal
TUser = record
  Name: RawUTF8;
  Password: RawUTF8;
  MobilePhoneNumber: RawUTF8;
  ID: TID;
end;
```

Here, we won't use TSQLRecord nor any other classes, just plain records, which will be used as neutral means of transmission. The difference between *Data Transfer Objects* and *business objects* or *Data Access Objects* (DAO) like our TSQLRecord is that a DTO does not have any behavior except for storage and retrieval of its own data. It can also be independent to the persistence layer, as implemented underneath our business domain. Using a record in *Delphi* ensure it won't be part of a complex business logic, but will remain used as value objects.

Now, let's come back to our TLoginController class. Here is the method we want to test:

```pascal
procedure TLoginController.ForgotMyPassword(const UserName: RawUTF8);
var
  U: TUser;
begin
  U := fUserRepository.GetUserByName(UserName);
  U.Password := Int32ToUtf8(Random(MaxInt));
  if fSmsSender.Send('Your new password is ' + U.Password, U.MobilePhoneNumber) then
    fUserRepository.Save(U);
end;
```

It will retrieve a TUser instance from its repository, then compute a new password, and send it via SMS to the user's mobile phone. On success, it is supposed to persist (save) the new user information to the database.

### 15.4.2. Why use fake / emulated interfaces?

Using the real implementation of IUserRepository will expect a true database to be available, with some potential issues on existing data. Similarly, the class implementing ISmsSender in the final project should better not be called during the test phase, since sending a SMS does cost money, and we will need a true mobile phone or Internet gateway to send the password.

For our testing purpose, we only want to ensure that when the "forgot my password" scenario is executed, the user record modification is persisted to the database.

One possibility could be to define two new dedicated classes, implementing both IUserRepository and ISmsSender interfaces. But it will be obviously time consuming and error-prone. This may be typical case when writing the test could be more complex than writing the method to be tested.

In order to maximize your ROI, and allow you to focus on your business logic, the *mORMot* framework proposes a simple and efficient way of creating "fake" implementations of any interface, just by defining the minimum behavior needed to run the test.

#### 15.4.2.1. Stubs and mocks

In the book "*The Art of Unit Testing*" (Osherove, Roy - 2009), a distinction is drawn between *stub* and *mock* objects:

- **Stubs** are the simpler of the two families of fake objects, simply implementing the same interface as
the object that they represent and returning pre-arranged responses. Thus a fake object merely provides a set of method stubs. Therefore the name. In mORMot, it is created via the TInterfaceStub generator;

- **Mocks** are described as a fake object that helps decide if a test failed or passed, by verifying if an interaction on an object occurred or not. Everything else is defined as a stub. In mORMot, it is created via the TInterfaceMock generator, which will link the fake object to an existing TSynTestCase instance - see below (page 626).

In practice, there should be only one mock per test, with as many stubs as necessary to let the test pass. Using a mocking/stubbing framework allows quick on-the-fly generation of interface with unique behavior dedicated to a particular test. In short, you define the stubs needed to let your test pass, and define one mock which will pass or fail the test depending on the feature you want to test.

Our mORMot framework follows this distinction, by defining two dedicated classes, named TInterfaceStub and TInterfaceMock, able to define easily the behavior of such classes.

### 15.4.2.2. Defining stubs

Let's implement our "forgot my password" scenario test.

The TSynTestCase child method could start as such:

```delphi
procedure TMyTest.ForgetThePassword;
var
  SmsSender: ISmsSender;
  UserRepository: IuserRepository;

This is all we need: one dedicated test case method, and our two local variables, ready to be set with our stubbed / mocked implementation classes.

First of all, we will need to implement ISmsSender.Send method. We should ensure that it returns true, to indicate a successful sending.

With mORMot, it is as simple as:

```delphi
TInterfaceStub.Create(TypeInfo(ISmsSender), SmsSender).
Returns('Send',[true]);
```

It will create a fake class (here called a "stub") emulating the whole ISmsSender interface, store it in the local SmsSender variable, and let its Send method return true.

What is nice with this subbing / mocking implementation is that:

- The "fluent" style of coding makes it easy to write and read the class behavior, without any actual coding in Delphi, nor class definition;
- Even if ISmsSender has a lot of methods, only Send matters for us: TInterfaceStub will create all those methods, and let them return default values, with additional line of code needed;
- Memory allocation will be handled by the framework: when SmsSender instance will be released, the associated TInterfaceStub data will also be freed (and in case a mock, any expectations will be verified).

### 15.4.2.3. Defining a mock

Now we will define another fake class, which may fail the test, so it is called a "mock", and the mORMot generator class will be TInterfaceMock:

```delphi
TInterfaceMock.Create(TypeInfo(IuserRepository), UserRepository, self).
ExpectsCount('Save', qoEqualTo, 1);
```

We provide the TMyTest instance as self to the TInterfaceMock constructor, to associate the
mocking aspects with this test case. That is, any registered Expects*() rule will let TMyTest.Check() be called with a boolean condition reflecting the test validation status of every rule.

The ExpectsCount() method is indeed where mocking is defined. When the UserRepository generated instance is released, TInterfaceMock will check all the Expects*() rules, and, in this case, check that the Save method has been called exactly one time (qoEqualTo,1).

15.4.2.4. Running the test

Since we have all the expected stub and mock at hand, let’s run the test itself:

```pascal
with TLoginController.Create(UserRepository,SmsSender) do
  try
    ForgotMyPassword('toto');
  finally
    Free;
  end;
```

That is, we run the actual implementation method, which will call our fake methods:

```pascal
procedure TLoginController.ForgotMyPassword(const UserName: RawUTF8);
var U: TUser;
begin
  U := fUserRepository.GetUserByName(UserName);
  U.Password := Int32ToUtf8(Random(MaxInt));
  if fSmsSender.Send('Your new password is '+U.Password,U.MobilePhoneNumber) then
    fUserRepository.Save(U);
end;
```

Let’s put all this together.
15.5. Stubs and Mocks in mORMot

Our mORMot framework is therefore able to stub or mock any Delphi interface.

We will now detail how it is expected to work.

15.5.1. Direct use of interface types without TypeInfo()

First of all, it is a good practice to always register your service interfaces in the unit which define their type, as such:

```pascal
unit MyServiceInterfaces;
...

type
  ISmsSender = interface (IInvokable)
  IUserRepository = interface (IInvokable)
...

initialization
  TInterfaceFactory.RegisterInterfaces(
    TypeInfo(ISmsSender), TypeInfo(IUserRepository));
end.
```

Then creating a stub or a mock could be done directly from the interface name, which will be transmitted as its TGUID, without the need of using the TypeInfo() pseudo-function:

```pascal
TInterfaceStub.Create(ISmsSender, SmsSender);
TInterfaceMock.Create(IUserRepository, UserRepository, self);
```

In the code below, we will assume that the interface type information has been registered, so that we may be able to use directly I* without the TypeInfo(I*) syntax.

15.5.2. Manual dependency injection

As usual, the best way to explain what a library does is to look at the code using it.

Here is an example (similar to the one shipped with RhinoMocks) of verifying that when we execute the "forgot my password" scenario as implemented by the TLoginController class, we actually called the Save() method:

```pascal
procedure TMyTest.ForgotMyPassword;
var
  SmsSender: ISmsSender;
  UserRepository: IUserRepository;
begin
  TInterfaceStub.Create(ISmsSender, SmsSender).
  Returns('Send',[true]);
  TInterfaceMock.Create(IUserRepository, UserRepository, self).
  ExpectsCount('Save', qoEqualTo, 1);
  with TLoginController.Create(UserRepository, SmsSender) do
  try
    ForgotMyPassword('toto');
  finally
    Free;
  end;
end;
```

And... that's all, since the verification will take place when IUserRepository instance will be released.

If you want to follow the "test spy" pattern (i.e. no expectation defined a priori, but manual check after the execution), you can use:
procedure TMyTest.ForgotMyPassword;
var
    SmsSender: ISmsSender;
    UserRepository: IUserRepository;
    Spy: TInterfaceMockSpy;
begin
    TInterfaceStub.Create(ISmsSender, SmsSender).
    Returns('Send', [true]);
    Spy := TInterfaceMockSpy.Create(IUserRepository, UserRepository, self);
    with TLoginController.Create(UserRepository, SmsSender) do
    try
        ForgotMyPassword('toto');
    finally
        Free;
    end;
    Spy.Verify('Save');
end;

This is something unique with our library: you can decide if you want to use the classic "expect-run-verify" pattern, or the somewhat more direct "run-verify" / "test spy" pattern. With mORMot, you pick up your mocking class (either TInterfaceMock or TInterfaceMockSpy), then use it as intended. You can even mix the two aspects in the same instance! It is just a matter of taste and opportunity for you to use the right pattern.

For another easier pattern, like the one in the Mockito home page:

TInterfaceMock.Create(ICalculator, ICalc, self).
    ExpectsCount('Multiply', qoEqualTo, 1).
    ExpectsCount('Add', [10, 20], qoEqualTo, 1);
ICalc.Add(10, 20);
ICalc.Multiply(10, 30)

If you want to follow the "test spy" pattern, you can use:

Mock := TInterfaceMockSpy.Create(ICalculator, ICalc, self);    
ICalc.Add(10, 20);    
ICalc.Multiply(10, 30)    
Mock.Verify('Add');    
Mock.Verify('Multiply', [10, 30]);

If you compare with existing mocking frameworks, even in other languages / platforms like the two above, you will find out that the features included in mORMot are quite complete:
- Stubbing of any method, returning default values for results;
- Definition of the stubbed behavior via a simple fluent interface, with TInterfaceStub.Returns(), including easy definition of returned results values, for the whole method or following parameters/arguments matchers;
- Handle methods with var, out or function result returned values - i.e. not only the function result (as other Delphi implementations does, due to a limitation of the TVirtualInterface standard implementation, on which mORMot does not rely), but all outgoing values, as an array of values;
- Stubbed methods can use delegates or event callbacks with TInterfaceStub.Executes() rule definitions, for the whole method or following parameters/arguments matchers, to run a more complex process;
- Stubbed methods can also raise exceptions with TInterfaceStub.Raises() rule definitions, for the whole method or following parameters/arguments matchers, if this is the behavior to be tested;
- Clear distinction between mocks and stubs, with two dedicated classes, named TInterfaceStub and TInterfaceMock;
- Mocks are directly linked to mORMot's unitary tests / test-driven classes - see below (page 626);
- Mocked methods can trigger test case failure with TInterfaceMock.Fails() definitions, for the whole method or following parameters/arguments matchers;
- Mocking via "expect-run-verify" or "run-verify" (aka "test spy") patterns, on choice, depending on
your testing expectations;
- Mocking validation against number of execution of a method, or a method with arguments/parameters matchers, or the global execution trace - in this case, pass count can be compared with operators like < <= = <> > >= and not only the classic exact-number-of-times and at-least-once verifications;
- Most common parameters and results can be defined as simple array of const in the Delphi code, or by supplying JSON arrays (needed e.g. for more complex structures like record values);
- Execution trace retrieval in easy to read or write text format (and not via complex "fluent" interface e.g. with when clauses);
- Auto-release of the TInterfaceStub TInterfaceMock TInterfaceMockSpy generator instance, when the interface is no longer required, to minimize the code to type, and avoid potential memory leaks;
- Works from Delphi 6 up to Delphi 10.3 Rio - since no use of syntax sugar like generics, nor the RTTI.pas features;
- Very good performance (the faster Delphi mocking framework, for sure), due to very low overhead and its reuse of mORMot's low-level interface-based services kernel using JSON serialization, which does not rely on the slow and limited TVirtualInterface.

15.5.3. Stubbing complex return values

Just imagine that the ForgotMyPassword method does perform an internal test:

```delphi
procedure TLoginController.ForgotMyPassword(const UserName: RawUTF8);
var U: TUser;
begin
  U := fUserRepository.GetUserByName(UserName);
  Assert(U.Name=UserName);
  U.Password := Int32ToUtf8(Random(MaxInt));
  if fSmsSender.Send('Your new password is '+U.Password,U.MobilePhoneNumber) then
    fUserRepository.Save(U);
end;
```

This will fail the test for sure, since by default, GetUserByName stubbed method will return a valid but void record. It means that U.Name will equal '', so the highlighted line will raise an EAssertionFailed exception.

Here is how we may enhance our stub, to ensure it will return a TUser value matching U.Name='toto':

```delphi
var UserRepository: IUserRepository;
U: TUser;
(...) U.Name := 'toto';
TInterfaceMock.Create(IUserRepository,UserRepository,self).Returns('GetUserByName''''toto'''' RecordSaveJSON(U,TypeInfo(TUser)));
ExpectsCount('Save',qoEqualTo,1);
```

The only trick in the above code is that we use RecordSaveJSON() function to compute the internal JSON representation of the record, as expected by mORMot's data marshalling.

15.5.4. Stubbing via a custom delegate or callback

In some cases, it could be very handy to define a complex process for a given method, without the need of writing a whole implementation class.

A delegate or event callback can be specified to implement this process, with three parameters marshalling modes:
- Via some Named[] variant properties (which are the default for the Ctxt callback parameter) - the easiest and safest to work with;
- Via some Input[] and Output[] variant properties;
- Directly as a JSON array text (the fastest, since native to the mORMot core).

Let's emulate the following behavior:

```pascal
function TServiceCalculator.Subtract(n1, n2: double): double;
begin
  result := n1 - n2;
end;
```

### 15.5.4.1. Delegate with named variant parameters

You can stub a method using a the Named[] variant arrays as such:

```pascal
TInterfaceStub.Create(ICalculator, ICalc).
  Executes('Subtract', IntSubtractVariant);
(...)
Check(ICalc.Subtract(10.5, 1.5) = 9);
```

The callback function can be defined as such:

```pascal
procedure TTestServiceOrientedArchitecture.IntSubtractVariant(
  Ctxt: TOnInterfaceStubExecuteParamsVariant);
begin
  Ctxt['result'] := Ctxt['n1'] - Ctxt['n2'];
end;
```

That is, callback shall use Ctxt[''] property to access the parameters and result as variant values.

In fact, we use the Ctxt.Named[] default property, so it is exactly as the following line:

```pascal
Ctxt.Named['result'] := Ctxt.Named['n1'] - Ctxt.Named['n2'];
```

If the execution fails, it shall execute Ctxt.Error() method with an associated error message to notify the stubbing process of such a failure.

Using named parameters has the advantage of being more explicit in case of change of the method signature (e.g. if you add or rename a parameter). It should be the preferred way of implementing such a callback, in most cases.

### 15.5.4.2. Delegate with indexed variant parameters

There is another way of implementing such a callback method, directly by using the Input[] and Output[] indexed properties. It should be (a bit) faster to execute:

```pascal
procedure TTestServiceOrientedArchitecture.IntSubtractVariant(
  Ctxt: TOnInterfaceStubExecuteParamsVariant);
begin
  with Ctxt do
  begin
    Output[0] := Input[0] - Input[1]; // result := n1-n2
  end;
```

Just as with TOnInterfaceStubExecuteParamsJSON implementation, Input[] index follows the exact order of const and var parameters at method call, and Output[] index follows the exact order of var and out parameters plus any function result.

That is, if you call:

```pascal
function Subtract(n1, n2: double): double;
...
MyStub.Subtract(100, 20);```
15.5.4.3. Delegate with JSON parameters

You can stub a method using a JSON array as such:

```pascal
TInterfaceStub.Create(ICalculator,ICalc).
Executes('Subtract',IntSubtractJSON);
(...)
Check(ICalc.Substract(10.5,1.5)=9);
```

The callback shall be defined as such:

```pascal
procedure TTestServiceOrientedArchitecture.IntSubtractJSON(
  Ctxt: TOnInterfaceStubExecuteParamsJSON);
var
  P: PUTF8Char;
begin
  // result := n1 - n2
  P := pointer(Ctxt.Params);
  Ctxt.Result := '[' + DoubleToStr(GetNextItemDouble(P) - GetNextItemDouble(P)) + ']';
end;
```

That is, it shall parse incoming parameters from Ctxt.Params, and store the result values as a JSON array in Ctxt.Result.

Input parameter order in Ctxt.Params follows the exact order of const and var parameters at method call, and output parameter order in Ctxt.Returns([ ]) or Ctxt.Result follows the exact order of var and out parameters plus any function result.

This method could have been written as such, if you prefer to return directly the JSON array:

```pascal
procedure TTestServiceOrientedArchitecture.IntSubtractJSON(
  Ctxt: TOnInterfaceStubExecuteParamsJSON);
var
  P: PUTF8Char;
begin
  // result := n1 - n2
  P := pointer(Ctxt.Params);
  Ctxt.Result := '[' + DoubleToStr(GetNextItemDouble(P) - GetNextItemDouble(P)) + ']';
end;
```

This may sound somewhat convenient here in case of double values, but it will be error prone if types are more complex. In all cases, using Ctxt.Returns([ ]) is the preferred method.

15.5.4.4. Accessing the test case when mocking

In case of mocking, you may add additional verifications within the implementation callback, as such:

```pascal
TInterfaceMock.Create(ICalculator,ICalc,self).
Executes('Subtract',IntSubtractVariant,'toto');
(...)
```

```pascal
procedure TTestServiceOrientedArchitecture.IntSubtractVariant(
  Ctxt: TOnInterfaceStubExecuteParamsVariant);
```

you have in TOnInterfaceStubExecuteParamsJSON:

```plaintext
Ctxt.Params = '100,20.5'; // at method call
Ctxt.Result = '[79.5]'; // after Ctxt.Returns([..])
```

and in the variant arrays:

```plaintext
Ctxt.Input[0] = 100; // =n1 at method call
Ctxt.Input[1] = 20.5; // =n2 at method call
Ctxt.Output[0] = 79.5; // =result after method call
```

In case of additional var or out parameters, those should be added to the Output[] array before the last one, which is always the function result.

If the method is defined as a procedure and not as a function, of course there is no last Output[] item, but only var or out parameters.
begin
  Ctxt.TestCase.Check(Ctxt.EventParams='toto');
  Ctxt['result'] := Ctxt['n1']-Ctxt['n2'];
end;

Here, an additional callback-private parameter containing 'toto' has been specified at TInterfaceMock definition. Then its content is checked on the associated test case via Ctxt.Sender instance. If the caller is not a TInterfaceMock, it will raise an exception when accessing the Ctxt.TestCase property.

### 15.5.5. Calls tracing

As stated above, mORMot is able to log all interface calls into its internal TInterfaceStub's structures. This is indeed the root feature of its "test spy" TInterfaceMockSpy.Verify() methods.

```pascal
Stub := TInterfaceStub.Create(ICalculator,I).
  SetOptions([imoLogMethodCallsAndResults]);
  Check(I.Add(10,20)=0,'Default result');
  Check(Stub.LogAsText='Add(10,20)=[0]');
```

Here above, we retrieved the whole call stack, including input parameters and returned results, as an easy-to-read JSON content. We found out that JSON is a very convenient way of tracing the method calls, both efficient for the computer and the human being hardly testing the code.

A more complex trace verification could be defined for instance, in the context of an interface `mock`:

```pascal
TInterfaceMock.Create(ICalculator,I,self).
  Returns('Add','30').
  Returns('Multiply',[60]).
  Returns('Multiply',[2,35],[70]).
  ExpectsCount('Multiply',qoEqualTo,2).
  ExpectsCount('Subtract',qoGreaterThan,0).
  ExpectsCount('ToTextFunc',qoLessThan,2).
  // check trace for a whole method execution
  ExpectsTrace('Add','Add(10,30)=[30]').;
  ExpectsTrace('Multiply','Multiply(10,30)=[60],Multiply(2,35)=[70]');
  // check trace for a whole method execution, filtering with given parameters
  ExpectsTrace('Multiply',[10,30],'Multiply(10,30)=[60]').;
  // check trace for the whole interface execution
  ExpectsTrace('Add(10,30)=[30],Multiply(10,30)=[60],Multiply(2,35)=[70],Subtract(2.3,1.2)=0,ToTextFunc(2.3)="default"');
  Returns('ToTextFunc','["default"]');
  Check(I.Add(10,30)=30);
  Check(I.Multiply(10,30)=60);
  Check(I.Multiply(2,35)=70);
  Check(I.Subtract(2.3,1.2)=0,'Default result');
  Check(I.ToTextFunc(2.3)='default');
```

The overloaded `ExpectsTrace()` methods are able to add some checks not only about the number of calls of a given method, but the exact order of the executed commands, with associated parameters and all retrieved result values. They can validate the trace of one specific method (optionally with a filter against the incoming parameters), or globally for the whole mocked interface.

Note that internally, those methods will compute a `Hash32()` hash value of the expected trace, which is a good way of minimizing data in memory or re-use a value retrieved at execution time for further regression testing. Some overloaded signatures are indeed available to directly specify the expected `Hash32()` value, in case of huge regression scenarios: run the test once, debugging all expected behavior by hand, then store the hash value to ensure that no expected step will be broken in the future.

You have even a full access to the internal execution trace, via the two TInterfaceStub.Log and LogCount properties. This will allow any validation of mocked interface calls logic, beyond
ExpectsTrace() possibilities.

You can take a look at TTestServiceOrientedArchitecture.MocksAndStubs regression tests, for a whole coverage of all the internal features.
15.6. Dependency Injection and Interface Resolution

In our example, we injected the dependencies explicitly as parameters to the class constructor - see Dependency Injection at constructors (page 406). We will present below (page 419), in a dedicated chapter, how the framework SOA features do resolve services as interfaces.

But real-world application may be much complex, and a generic way of resolving dependencies, and Inversion Of Control (aka IoC) has been implemented.

First of all, if you inherit from TInjectableObject, you will be able to resolve dependencies in two ways:
- Explicitly via its Resolve() overloaded methods, for lazy initialization of any registered interface;
- Automatically at instance creation, for all its published properties declared with an interface type.

A dedicated set of overloaded constructors is also available at TInjectableObject class level, so that you may be able to easily stub/mock or inject any instance, e.g. for testing purposes:

```pascal
procedure TMyTestCase.OneTestCaseMethod;
var
  Test: IServiceToBeTested;
begin
  Test := TServiceToBeTested.CreateInjected(
    [ICalculator],
    [TInterfaceMock.Create(IPersistence,self).
      ExpectsCount('SaveItem',qoEqualTo,1),
      RestInstanceId.Services],
    [AnyInterfacedObject]);
...```

In this code, we have Direct use of interface types without TypeInfo() (page 410). So we could write directly ICalculator or IPersistence to refer to an explicit interface type.

This test case (TMyTestCase inherits from TSynTestCase) will create a TServiceToBeTested instance, create a TInterfaceStub for its ICalculator dependency, then a TInterfaceMock expecting the IPersistence.SaveItem method to be called exactly one time, allowing resolution from a TSQLRest.Services SOA resolver, and injecting a pre-existing AnyInterfacedObject TInterfacedObject instance.

Then, dependency resolution may take place as published properties:

```pascal
type
  TServiceToBeTested = class(TInjectableObject)
    protected
      fCalculator: ICalculator;
    ...;
    published
      property Calculator: ICalculator read fCalculator;
    ...;
  end;
...```

```pascal
function TServiceToBeTested.DoCalculation(a,b: integer): integer;
begin
  result := Calculator.Add(a,b);
end;
```

This fCalculator instance will be resolved and instantiated by TInjectableObject.Create, then released as any regular interface field in the class destructor. You do not have to overload the TServiceToBeTested constructor, nor manage this fCalculator life time. Its auto-created instance...
will be shared by the whole TServiceToBeTested context, so it should be either stateless (like adding two numbers), or expected to evolve at each use.

Sometimes, there may be an over-cost to initialize such properties each time a TServiceToBeTested class instance is created. Or maybe the interface implementation is not stateless, and a new instance should be retrieved before each use. As an alternative, any interface may be resolved on need, in a lazy way:

```pascal
procedure TServiceToBeTested.DoSomething;
var persist: IPersistence;
begin
  Resolve(IPersistence,persist);
  persist.SaveItem('John','Doe');
end;
```

The TInjectableObject.Resolve() overloaded methods will retrieve one instance of the asked interface. The above code will raise an exception if the supplied IPersistence was not previously registered to the TInjectableObject class.

When such an TInjectableObject instance is created within mORMot's SOA methods (i.e. TSQLRest.Services property), the injection will transparently involve all registered classes. Also take a look at the TInterfaceResolverInjected.RegisterGlobal() overloaded methods, which are able to register some class types or instances globally for the whole executable context. Just make sure that you won't break the Open/Closed Principle (page 393), by defining such a global registration, which should occur only for specific needs, truly orthogonal to the whole application, or specific to a test case.
16. Client-Server services via interfaces

In real world, especially when your application relies heavily on services, the Client-Server services via methods (page 373) implementation pattern has some drawbacks:

- Most content marshalling is to be done by hand, so may introduce implementation issues;
- Client and server side code does not have the same implementation pattern, so you will have to code explicitly data marshalling twice, for both client and server (DataSnap and WCF both suffer from a similar issue, by which client classes shall be coded separately, most time generated by a Wizard);
- You can not easily test your services, unless you write a lot of code to emulate a "fake" service implementation;
- The services do not have any hierarchy, and are listed as a plain list, which is not very convenient;
- It is difficult to synchronize several service calls within a single context, e.g. when a workflow is to be handled during the application process (you have to code some kind of state machine on both sides, and define all session handling by hand);
- Security is handled globally for the user, or should be checked by hand in the implementation method (using the Ctxt.Session* members);
- There is no way of implementing service callbacks, using e.g. WebSockets.

You can get rid of those limitations with the interface-based service implementation of mORMot. For a detailed introduction and best practice guide to SOA, see Service-Oriented Architecture (SOA) (page 90). All commonly expected SOA features are now available in the current implementation of the mORMot framework (including service catalog aka "broker", via the optional publication of interface signatures).
### 16.1. Implemented features

Here are the key features of the current implementation of services using interfaces in the *Synopses mORMot framework*, as implemented in `mORMot.pas` unit:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Orientation</td>
<td>Allow loosely-coupled relationship</td>
</tr>
<tr>
<td>Design by contract</td>
<td>Service Contracts are defined in Delphi code as standard interface</td>
</tr>
<tr>
<td></td>
<td>custom types</td>
</tr>
<tr>
<td>Factory driven</td>
<td>Get an implementation instance from a given interface</td>
</tr>
<tr>
<td>Server factory</td>
<td>You can get an implementation on the server side</td>
</tr>
<tr>
<td>Client factory</td>
<td>You can get a &quot;fake&quot; implementation on the client side, remotely calling</td>
</tr>
<tr>
<td></td>
<td>the server to execute the process</td>
</tr>
<tr>
<td>Cross-platform clients</td>
<td>A mORMot server is able to generate cross-platform client code via a set</td>
</tr>
<tr>
<td></td>
<td>of templates - see below (page 481)</td>
</tr>
<tr>
<td>Auto marshalling</td>
<td>The contract is transparently implemented: no additional code is needed</td>
</tr>
<tr>
<td></td>
<td>e.g. on the client side, and will handle simple types (strings, numbers,</td>
</tr>
<tr>
<td></td>
<td>dates, sets and enumerations) and high-level types (objects, collections,</td>
</tr>
<tr>
<td></td>
<td>records, dynamic arrays, variants) from Delphi 6 up to Delphi 10.3 Rio</td>
</tr>
<tr>
<td>Flexible</td>
<td>Methods accept per-value or per-reference parameters</td>
</tr>
<tr>
<td>Instance lifetime</td>
<td>An implementation class can be:</td>
</tr>
<tr>
<td></td>
<td>- Created on every call,</td>
</tr>
<tr>
<td></td>
<td>- Shared among all calls,</td>
</tr>
<tr>
<td></td>
<td>- Shared for a particular user or group,</td>
</tr>
<tr>
<td></td>
<td>- Dedicated to the thread it runs on,</td>
</tr>
<tr>
<td></td>
<td>- Alive as long as the client-side interface is not released,</td>
</tr>
<tr>
<td></td>
<td>- Or as long as an authentication session exists</td>
</tr>
<tr>
<td>Stateless</td>
<td>Following a standard request/reply pattern</td>
</tr>
<tr>
<td>Statefull</td>
<td>Server side implementation may be synchronized with client-side</td>
</tr>
<tr>
<td></td>
<td>interface, e.g. over WebSockets</td>
</tr>
<tr>
<td>Dual way</td>
<td>You can define callbacks, using e.g. WebSockets for immediate</td>
</tr>
<tr>
<td></td>
<td>notification</td>
</tr>
<tr>
<td>Signed</td>
<td>The contract is checked to be consistent before any remote execution</td>
</tr>
<tr>
<td>Secure</td>
<td>Every service and/or methods can be enabled or disabled on need</td>
</tr>
<tr>
<td>Safe</td>
<td>Using extended RESTful authentication - see below (page 546)</td>
</tr>
<tr>
<td>Multi-hosted (with DMZ)</td>
<td>Services are hosted by default within the main ORM server, but can have</td>
</tr>
<tr>
<td></td>
<td>their own process, with a dedicated connection to the ORM core</td>
</tr>
<tr>
<td>Broker ready</td>
<td>Service meta-data can be optionally revealed by the server</td>
</tr>
</tbody>
</table>
### Multiple transports

All Client-Server protocols of mORMot are available, i.e. direct in-process connection, Windows Messages, named pipes, TCP/IP-HTTP

### JSON based

Transmitted data uses *JavaScript Object Notation*

### Routing choice

Services are identified either at the URI level (the RESTful way), or in a JSON-RPC model (the AJAX way), or via any custom format (using class inheritance)

### AJAX and RESTful

JSON and HTTP combination allows services to be consumed from AJAX rich clients

### Light & fast

Performance and memory consumption are very optimized, in order to ensure scalability and ROI

### 16.2. How to make services

The typical basic tasks to perform are the following:
- Define the service contract;
- Implement the contract;
- Configure and host the service;
- Build a client application.

We will describe those items.
16.3. Defining a service contract

In a SOA, services tend to create a huge list of operations. In order to facilitate implementation and maintenance, operations shall be grouped within common services.

Before defining how such services are defined within mORMot, it is worth applying the Service-Oriented Architecture (SOA) (page 90) main principles, i.e. loosely-coupled relationship. When you define mORMot contracts, ensure that this contract will stay un-coupled with other contracts. It will help writing SOLID code, enhance maintainability, and allow introducing other service providers on demand (some day or later, you'll certainly be asked to replace one of your service with a third-party existing implementation of the corresponding feature: you shall at least ensure that your own implementation will be easily re-coded with external code, using e.g. a SOAP/WSDL gateway).

16.3.1. Define an interface

The service contract is to be defined as a plain Delphi interface type. In fact, the sample type as stated above - see Interfaces (page 385) - can be used directly:

```delphi
type
  ICalculator = interface(IInvokable)
    ['{9A68CB8ED-C8B2-4E09-87D4-4A16F496E5FE}']
    // add two signed 32-bit integers
    function Add(n1, n2: integer): integer;
  end;
```

This ICalculator.Add method will define one "Add" operation, under the "ICalculator" service (which will be named internally 'Calculator' by convention). This operation will expect two numbers as input, and then return the sum of those numbers.

The current implementation of service has the following expectations:
- Any interface inheriting from IInvokable, with a GUID, can be used - we expect the RTTI to be available, so IInvokable is a good parent type;
- You can inherit an interface from an existing one: in this case, the inherited methods will be part of the child interface, and will be expected to be implemented (just as with standard Delphi code);
- Only plain ASCII names are allowed for the type definition (as it is conventional to use English spelling for service and operation naming);
- Calling convention shall be register (the Delphi's default) - nor stdcall nor cdecl is available yet, but this won't be a restriction since the interface definition is dedicated to Delphi code scope;
- Methods can have a result, and accept per-value or per-reference parameters.

In fact, parameters expectations are the following:
- Simple types (strings, numbers, dates, sets and enumerations) and high-level types (objects, collections, records and dynamic arrays) are handled - see below for the details;
- They can be defined as const, var or out - in fact, const and var parameters values will be sent from the client to the server as JSON, and var and out parameters values will be returned as JSON from the server;
- procedure or function kind of method definition are allowed;
- Only exception is that you can’t have a function returning a class instance (how will know when to release the instance in this case?), but such instances can be passed as const, var or out parameters (and published properties will be serialized within the JSON message);
- In fact, the TCollection kind of parameter is not directly handled by the framework: you shall define a TInterfacedCollection class, overriding its GetClass abstract virtual method (otherwise the server side won't be able to create the kind of collection as expected);
Special TServiceCustomAnswer kind of record can be used as function result to specify a custom content (with specified encoding, to be used e.g. for AJAX or HTML consumers) - in this case, no var nor out parameters values shall be defined in the method (only the BLOB value is returned).

16.3.2. Service Methods Parameters

Handled types of parameters are:

<table>
<thead>
<tr>
<th>Delphi type</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>Transmitted as JSON true/false</td>
</tr>
<tr>
<td>integer</td>
<td>Transmitted as JSON numbers</td>
</tr>
<tr>
<td>cardinal Int6</td>
<td>Transmitted as JSON numbers</td>
</tr>
<tr>
<td>double</td>
<td>Transmitted as JSON numbers</td>
</tr>
<tr>
<td>currency</td>
<td>Transmitted as JSON numbers</td>
</tr>
<tr>
<td>enumerations</td>
<td>Transmitted as JSON number</td>
</tr>
<tr>
<td>set</td>
<td>Transmitted as JSON number - one bit per element (up to 32 elements)</td>
</tr>
<tr>
<td>TDateTime</td>
<td>Transmitted as ISO 8601 JSON text</td>
</tr>
<tr>
<td>TDateTimeMS</td>
<td>Transmitted as ISO 8601 JSON text</td>
</tr>
<tr>
<td>WideString</td>
<td>Transmitted as UTF-8 JSON text (UTF-8 encoded)</td>
</tr>
<tr>
<td>SynUnicode</td>
<td>Transmitted as UTF-8 JSON text, but prior to Delphi 2009, the framework will ensure that both client and server sides use the same ANSI code page - so you should better use RawUTF8 everywhere</td>
</tr>
<tr>
<td>string</td>
<td>UTF-8 buffer transmitted with no serialization (wheras a RawUTF8 will be escaped as a JSON string) - expects to contain valid JSON content, e.g. for TSQLTableJSON requests</td>
</tr>
<tr>
<td>RawJSON</td>
<td>Transmitted as Base64 encoded JSON text - see below (page 469) for optional binary transmission</td>
</tr>
<tr>
<td>RawByteString</td>
<td>Transmitted as Base64 encoded JSON text - see below (page 469) for optional binary transmission</td>
</tr>
<tr>
<td>TPersistent</td>
<td>Published properties will be transmitted as JSON object</td>
</tr>
<tr>
<td>TSQLRecord</td>
<td>All published fields (including ID) will be transmitted as JSON object</td>
</tr>
<tr>
<td>TCollection</td>
<td>Not allowed directly: inherit from TInterfacedCollection or call TJSONSerializer. RegisterCollectionForJSON()</td>
</tr>
<tr>
<td>TInterfacedCollection</td>
<td>Transmitted as a JSON array of JSON objects - see below (page 426)</td>
</tr>
<tr>
<td>TObjectList</td>
<td>Transmitted as a JSON array of JSON objects, with a &quot;ClassName&quot;: &quot;TMyClass&quot; field to identify the type - see TObjectList serialization (page 309)</td>
</tr>
<tr>
<td>any TObject</td>
<td>See TObject serialization (page 306)</td>
</tr>
<tr>
<td>dynamic arrays</td>
<td>Transmitted as JSON arrays - see TDynArray dynamic array wrapper (page 107)</td>
</tr>
</tbody>
</table>
Need to have RTTI (so a string or dynamic array field within), just like with regular Delphi interface expectations - transmitted as binary with Base64 encoding before Delphi 2010, or as JSON object thanks to the enhanced RTTI available since, or via an custom JSON serialization - see Record serialization (page 297)

Transmitted as JSON, with support of TDocVariant custom variant type (page 112) for objects and arrays; OLE variant arrays are not handled: use _Arr([]) _ArrFast([]) instead

If used as a function result (not as parameter), the supplied content will be transmitted directly to the client (with no JSON serialization); in this case, no var nor out parameters are allowed in the method - it will be compatible with both our TServiceFactoryClient implementation, and any other service consumers (e.g. AJAX)

A callback instance could be specified, to allow asynchronous notification, using e.g. WebSockets - see below (page 444)

You can therefore define complex interface types, as such:

```delphi
type
  ICalculator = interface(IInvokable)
  ['{9A68C8ED-CEB2-4E09-87D4-4A16F496E5FE}']
  /// add two signed 32-bit integers
  function Add(n1,n2: integer): integer;
  /// multiply two signed 64-bit integers
  function Multiply(n1,n2: Int64): Int64;
  /// subtract two floating-point values
  function Subtract(n1,n2: double): double;
  /// convert a currency value into text
  procedure ToText(Value: Currency; var Result: RawUTF8);
  /// convert a floating-point value into text
  function ToTextFunc(Value: double): string;
  /// do some work with strings, sets and enumerates parameters,
  /// testing also var (in/out) parameters and set as a function result
  function SpecialCall(Txt: RawUTF8; var Int: integer; var Card: cardinal; field: TSynTableFieldTypes;
    fields: TSynTableFieldTypes; var options: TSynTableFieldOptions): TSynTableFieldTypes;
  /// test integer, strings and wide strings dynamic arrays, together with records
  function ComplexCall(const Ints: TIntegerDynArray; Strs1: TRawUTF8DynArray;
    var Str2: TWideStringDynArray; const Rec1: TVirtualTableModuleProperties;
    var Rec2: TSQLRestCacheEntryValue): TSQLRestCacheEntryValue;
  /// test variant kind of parameters
  function TestVariants(const Text: RawUTF8; V1: variant; var V2: variant): variant;
  /// validates ArgsInputIsOctetStream raw binary upload
  function DirectCall(const Data: TSQLRawBlob): integer;
end;
```

Note how SpecialCall and ComplexCall methods have quite complex parameters definitions, including dynamic arrays, sets and records. DirectCall will use binary POST, by-passing Base64 JSON encoding - see below (page 469). The framework will handle const and var parameters as expected, i.e. as input/output parameters, also on the client side. Any simple types of dynamic arrays (like TIntegerDynArray, TRawUTF8DynArray, or TWideStringDynArray) will be serialized as plain JSON arrays - the framework is able to handle any dynamic array definition, but will serialize those simple types in a more AJAX compatible way, thanks to the enhanced RTTI available since to Delphi 2010.

16.3.3. TPersistent / TSQLRecord parameters
As stated above, *mORMot* does not allow a method function to return a class instance.

That is, you can't define such a method:

```plaintext
ICustomerFactory = interface(IInvokable)
    ['{7700009F-15F4-4307-B2AD-BBAE42FE70C0}']
    function NewCustomer: TCustomer;
end;
```

Who will be in charge of freeing the instance, in client-server mode? There is no standard allocation scheme, in *Delphi*, for such parameters. So every TObject parameter instance shall be managed by the caller, i.e. allocated before the call and released after it. The method will just read or write the instance published properties, and serialize them as JSON.

What you can define is such a method:

```plaintext
ICustomerFactory = interface(IInvokable)
    ['{7700009F-15F4-4307-B2AD-BBAE42FE70C0}']
    procedure NewCustomer(out aCustomer: TCustomer);
end;
```

Note that here the `out` keyword does not indicate how the memory is allocated, but shows the communication direction of the remote service, i.e. it will serialize the object at method return. The caller shall instantiate an instance before call - whereas for "normal" *Delphi* code, it may be up to the method to instantiate the instance, and return it.

Then your client code can use it as such:

```plaintext
var
    Factory: ICustomerFactory;
    Repository: ICustomerRepository;
    Customer: TCustomer;
...
Customer := TCustomer.Create; // client side manage object instance
try
    Customer.FirstName := StringToUTF8(EditFirstName.Text);
    Customer.LastName := StringToUTF8(EditLastName.Text);
    NewCutomerID := Repository.Save(Customer); // persist the object
finally
    Customer.Free; // properly manage memory
end;
```

Or, using both *Factory* and *Repository* patterns, as proposed by below (page 595):

```plaintext
var
    Factory: ICustomerFactory;
    Repository: ICustomerRepository;
    Customer: TCustomer;
...
Factory.NewCustomer(Customer); // get a new object instance
try
    Customer.FirstName := StringToUTF8(EditFirstName.Text);
    Customer.LastName := StringToUTF8(EditLastName.Text);
    NewCutomerID := Repository.Save(Customer); // persist the object
finally
    Customer.Free; // properly manage memory
end;
```

In real live, it may be very easy to wrongly write a server method returning an existing instance, which will be released by the server SOA caller, and will trigger unexpected A/V randomly - very difficult to track - on the server side. Which is what we want to avoid... Whereas a pointer to nil gives always a clear access violation on the client side, which doesn't affect the server.

So this requirement/limitation was designed as such to make the server side more resilient to errors, even if the client side is a bit more complex to work with. Usually, on the client side, you can safely pre-allocate your object instances, and reuse them.
16.3.4. Record parameters

By default, any record parameter or function result will be serialized with a proprietary binary (and optimized) layout, then transmitted as a JSON string, after Base64 encoding.

Even if older versions of Delphi are not able to generate the needed RTTI information for such serialization, allowing us only to use an efficient but proprietary binary layout, the mORMot framework offers a common way of implementing any custom serialization of records. See Record serialization (page 297).

Note that the callback signature used for records matches the one used for dynamic arrays serializations - see Dynamic array serialization (page 303) - as it will be shared between the two of them.

When records are used as Data Transfer Objects within services (which is a good idea in common SOA implementation patterns), such a custom serialization format can be handy, and makes more natural service consumption with AJAX clients.

16.3.5. TCollection parameters

16.3.5.1. Use of TCollection

With mORMot services, you are able to define such a contract, e.g. for a TCollTests collection of TCollTest items:

\[\text{procedure Collections(Item: TCollTest; var List: TCollTests; out Copy: TCollTests);}\]

Typical implementation of this contract may be:

\[\text{procedure TServiceComplexCalculator.Collections(Item: TCollTest); var List: TCollTests; out Copy: TCollTests); begin CopyObject(Item,List.Add); CopyObject(List,Copy); end;}\]

That is, it will append the supplied Item object to the provided List content, then return a copy in the Copy content:

- Setting Item without var or out specification is doing the same as const: it will be serialized from client to server (and not back from server to client);
- Setting List as var parameter will let this collection to be serialized from client to server, and back from server to the client;
- Setting Copy as out parameter will let this collection to be serialized only from server to client.

Note that const / var / out kind of parameters are used at the contract level in order to specify the direction of serialization, and not as usual (i.e. to define if it is passed by value or by reference). All class parameters shall be instantiated before method call: you can not pass any object parameter as nil (nor use it in a function result): it will raise an error.

Due to the current implementation pattern of the TCollection type in Delphi, it was not possible to implement directly this kind of parameter.

In fact, the TCollection constructor is defined as such:

\[\text{constructor Create(ItemClass: TCollectionItemClass);}\]

And, on the server side, we do not know which kind of TCollectionItemClass is to be passed. Therefore, the TServiceFactoryServer is unable to properly instantiate the object instances,
supplying the expected item class.

The framework propose two potential solutions:
- You can let your collection class inherit from the new TInterfacedCollection type;
- You can call the TJSONSerializer.RegisterCollectionForJSON() method to register the collection type and its associated item class.

We will now describe both ways.

16.3.5.2. Inherit from TInterfacedCollection

A dedicated TInterfacedCollection abstract type has been defined:

```pascal
TInterfacedCollection = class(TCollection)
  protected
  class function GetClass: TCollectionItemClass; virtual; abstract;
  public
  constructor Create; reintroduce; virtual;
end;
```

In order to use a collection of objects, you will have to define at least the abstract method, for instance:

```pascal
TCollTests = class(TInterfacedCollection)
  protected
  class function GetClass: TCollectionItemClass; override;
end;
```

```pascal
class function TCollTests.GetClass: TCollectionItemClass;
begin
  result := TCollTest;
end;
```

Or, if you want a more complete / convenient implementation:

```pascal
TCollTests = class(TInterfacedCollection)
  private
    function GetCollItem(Index: Integer): TCollTest;
  protected
    class function GetClass: TCollectionItemClass; override;
  public
    function Add: TCollTest;
    property Item[Index: Integer]: TCollTest read GetCollItem; default;
end;
```

All other methods and properties (like GetColItem / Add / Items[]) are to be defined as usual.

16.3.5.3. Register a TCollection type

The other way of using TCollection kind of parameters is to declare it explicitly to the framework. You should call JSONSerializer.RegisterCollectionForJSON() with the corresponding TCollection / TCollectionItem class type pair.

Consider a dedicated class:

```pascal
TMyCollection = type(TCollection)
```

Note that a dedicated type is needed here. You just can’t use this registration over a plain TCollection.

Then, for instance, after calling:

```pascal
TJSONSerializer.RegisterCollectionForJSON(TMyCollection, TMyCollectionItem);
```
The following lines of code are the same:

```delphi
MyColl := TMyCollection.Create(TMyCollectionItem);
MyColl := ClassInstanceCreate(TMyCollection) as TMyCollection;
MyColl := ClassInstanceCreate('TMyCollection') as TMyCollection;
```

The last two will retrieve the associated TMyCollectionItem class type from the previous registration.

Thanks to this internal registration table, mORMot will be able to serialize and unserialize plain TCollection type.
16.4. Server side

16.4.1. Implementing the service contract

In order to have an operating service, you'll need to implement a Delphi class which matches the expected interface.

In fact, the sample type as stated above - see Interfaces (page 385) - can be used directly:

```
type
  TServiceCalculator = class(TInterfacedObject, ICalculator)
  public
    function Add(n1, n2: integer): integer;
  end;

function TServiceCalculator.Add(n1, n2: integer): integer;
begin
  result := n1+n2;
end;
```

And... That is all we need. The Delphi IDE will check at compile time that the class really implements the specified interface definition, so you'll be sure that your code meets the service contract expectations. Exact match (like handling type of parameters) will be checked by the framework when the service factory will be initialized, so you won't face any runtime exception due to a wrong definition.

Here the class inherits from TInterfacedObject, but you could use any plain Delphi class: the only condition is that it implements the ICalculator interface.

16.4.2. Set up the Server factory

In order to have a working service, you'll need to initialize a server-side factory, as such:

```
Server.ServiceRegister(TServiceCalculator,[TypeInfo(ICalculator)],sicShared);
```

You may prefer Direct use of interface types without TypeInfo() (page 410), if the type has previously been registered:

```
Server.ServiceDefine(TServiceCalculator,[ICalculator],sicShared);
```

The Server instance can be any TSQLRestServer inherited class, implementing any of the supported protocol of mORMot's Client-Server process (page 318), embedding a full SQLite3 engine (i.e. a TSQLRestServerDB class) or a lighter in-memory engine (i.e. a TSQLRestServerFullMemory class - which is enough for hosting services with authentication).

The code line above will register the TServiceCalculator class to implement the ICalculator service, with a single shared instance life time (specified via thesicShared parameter). An optional time out value can be specified, in order to automatically release a deprecated instance after some inactivity.

Whenever a service is executed, an implementation class is to be available. The life time of this implementation class is defined on both client and server side, by specifying a TServiceInstanceImplementation value. This setting must be the same on both client and server sides (it will be checked by the framework).

16.4.3. Instances life time implementation
The available instance management options are the following:

<table>
<thead>
<tr>
<th>Lifetime</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sicSingle</td>
<td>One class instance is created per call:</td>
</tr>
<tr>
<td></td>
<td>- This is the most expensive way of implementing the service, but is safe for simple workflows (like a one-type call);</td>
</tr>
<tr>
<td></td>
<td>- This is the default setting for TSQLRestServer.ServiceRegister/ServiceDefine methods.</td>
</tr>
<tr>
<td>sicShared</td>
<td>One object instance is used for all incoming calls and is not recycled subsequent to the calls.</td>
</tr>
<tr>
<td>sicClientDriven</td>
<td>One object instance will be created in synchronization with the client-side lifetime of the corresponding interface: when the interface will be released on client (either when it comes out of scope or set to nil), it will be released on the server side - a numerical identifier will be transmitted with all JSON requests</td>
</tr>
<tr>
<td>sicPerSession</td>
<td>One object instance will be maintained during the whole running session.</td>
</tr>
<tr>
<td>sicPerUser</td>
<td>One object instance will be maintained and associated with the running user.</td>
</tr>
<tr>
<td>sicPerGroup</td>
<td>One object instance will be maintained and associated with the running user's authorization group.</td>
</tr>
<tr>
<td>sicPerThread</td>
<td>One object instance will be maintained and associated with the running thread.</td>
</tr>
</tbody>
</table>

Of course, sicPerSession, sicPerUser and sicPerGroup modes will expect a specific user to be authenticated. Those implementation patterns will therefore only be available if the RESTful authentication is enabled between client and server.

Typical use of each mode may be the following:

<table>
<thead>
<tr>
<th>Lifetime</th>
<th>Use case</th>
</tr>
</thead>
<tbody>
<tr>
<td>sicSingle</td>
<td>An asynchronous process (may be resource consuming)</td>
</tr>
<tr>
<td>sicShared</td>
<td>Either a very simple process, or requiring some global data</td>
</tr>
<tr>
<td>sicClientDriven</td>
<td>The best candidate to implement a Business Logic workflow</td>
</tr>
<tr>
<td>sicPerSession</td>
<td>To maintain some data specific to the client application</td>
</tr>
<tr>
<td>sicPerUser</td>
<td>Access to some data specific to one user</td>
</tr>
<tr>
<td>sicPerGroup</td>
<td>Access to some data shared by a user category (e.g. administrators, or guests)</td>
</tr>
<tr>
<td>sicPerThread</td>
<td>Thread-oriented process (e.g. for proper library initialization)</td>
</tr>
</tbody>
</table>

In the current implementation of the framework, the class instance is allocated in memory.

This has two consequences:
- In client-server architecture, it is very likely that a lot of such instances will be created. It is therefore mandatory that it won't consume a lot of resource, especially with long-term life time: e.g. you should not store any BLOB within these instances, but try to restrict the memory use to the minimum. For a more consuming operation (a process which may need memory and CPU power), the sicSingle mode is preferred.

- There is no built-in data durability yet: service implementation shall ensure that data remaining in memory between calls (i.e. when not defined in sicSingle mode) won't be missing in case of server shutdown. It is up to the class to persist the needed data - using e.g. Object-Relational Mapping (page 130).

Note also that all those life-time modes expect the method implementation code to be thread-safe and reintrant on the server side - only exceptions are sicSingle mode, which will have its own running instance, and sicPerThread, which will have its methods always run in the same thread context. In practice, the same user can open more than one connection, therefore it is recommended to protect all implementation class method process, or set the execution options as expected - see below (page 461).

In order to illustrate sicClientDriven implementation mode, let's introduce the following interface and its implementation (extracted from the supplied regression tests of the framework):

```pascal
type
  IComplexNumber = interface(IInvokable)
  [{29D753B2-E7EF-41B3-87C3-827FEB082DC1}]
    procedure Assign(aReal, aImaginary: double);
    function GetImaginary: double;
    function GetReal: double;
    procedure SetImaginary(const Value: double);
    procedure SetReal(const Value: double);
    procedure Add(aReal, aImaginary: double);
  property Real: double read GetReal write SetReal;
  property Imaginary: double read GetImaginary write SetImaginary;
end;
```

Purpose of this interface is to store a complex number within its internal fields, then retrieve their values, and define a "Add" method, to perform an addition operation. We used properties, with associated getter and setter methods, to provide object-like behavior on Real and Imaginary fields, in the code.

This interface is implemented on the server side by the following class:

```pascal
type
  TServiceComplexNumber = class(TInterfacedObject, IComplexNumber)
  private
    fReal: double;
    fImaginary: double;
    function GetImaginary: double;
    function GetReal: double;
    procedure SetImaginary(const Value: double);
    procedure SetReal(const Value: double);
  public
    procedure Assign(aReal, aImaginary: double);
    procedure Add(aReal, aImaginary: double);
    property Real: double read GetReal write SetReal;
    property Imaginary: double read GetImaginary write SetImaginary;
end;

{ TServiceComplexNumber }

procedure TServiceComplexNumber.Add(aReal, aImaginary: double);
begin
  fReal := fReal+aReal;
```

- In client-server architecture, it is very likely that a lot of such instances will be created. It is therefore mandatory that it won't consume a lot of resource, especially with long-term life time: e.g. you should not store any BLOB within these instances, but try to restrict the memory use to the minimum. For a more consuming operation (a process which may need memory and CPU power), the sicSingle mode is preferred.

- There is no built-in data durability yet: service implementation shall ensure that data remaining in memory between calls (i.e. when not defined in sicSingle mode) won't be missing in case of server shutdown. It is up to the class to persist the needed data - using e.g. Object-Relational Mapping (page 130).

Note also that all those life-time modes expect the method implementation code to be thread-safe and reintrant on the server side - only exceptions are sicSingle mode, which will have its own running instance, and sicPerThread, which will have its methods always run in the same thread context. In practice, the same user can open more than one connection, therefore it is recommended to protect all implementation class method process, or set the execution options as expected - see below (page 461).

In order to illustrate sicClientDriven implementation mode, let's introduce the following interface and its implementation (extracted from the supplied regression tests of the framework):

```pascal
type
  IComplexNumber = interface(IInvokable)
  [{29D753B2-E7EF-41B3-87C3-827FEB082DC1}]
    procedure Assign(aReal, aImaginary: double);
    function GetImaginary: double;
    function GetReal: double;
    procedure SetImaginary(const Value: double);
    procedure SetReal(const Value: double);
    procedure Add(aReal, aImaginary: double);
  property Real: double read GetReal write SetReal;
  property Imaginary: double read GetImaginary write SetImaginary;
end;
```

Purpose of this interface is to store a complex number within its internal fields, then retrieve their values, and define a "Add" method, to perform an addition operation. We used properties, with associated getter and setter methods, to provide object-like behavior on Real and Imaginary fields, in the code.

This interface is implemented on the server side by the following class:

```pascal
type
  TServiceComplexNumber = class(TInterfacedObject, IComplexNumber)
  private
    fReal: double;
    fImaginary: double;
    function GetImaginary: double;
    function GetReal: double;
    procedure SetImaginary(const Value: double);
    procedure SetReal(const Value: double);
  public
    procedure Assign(aReal, aImaginary: double);
    procedure Add(aReal, aImaginary: double);
    property Real: double read GetReal write SetReal;
    property Imaginary: double read GetImaginary write SetImaginary;
end;

{ TServiceComplexNumber }

procedure TServiceComplexNumber.Add(aReal, aImaginary: double);
begin
  fReal := fReal+aReal;
```
function TServiceComplexNumber.GetReal: double;
begin
result := fReal;
end;

procedure TServiceComplexNumber.SetReal(const Value: double);
begin
fReal := Value;
end;

This interface is registered on the server side as such:

Server.ServiceDefine(TServiceComplexNumber,[IComplexNumber],sicClientDriven);

Using the sicClientDriven mode, also the client side will be able to have its own life time handled as expected. That is, both fReal and fImaginary field will remain allocated on the server side as long as needed. A time-out driven garbage collector will delete any un-closed pending session, therefore release resources allocated in sicClientDriven mode, even in case of a broken connection.

16.4.4. Accessing low-level execution context

16.4.4.1. Retrieve information from the global ServiceContext

When any interface-based service is executed, a global threadvar named ServiceContext can be accessed to retrieve the currently running context on the server side.

You will have access to the following information, which could be useful for sicPerSession, sicPerUser and sicPerGroup instance life time modes:

TServiceRunningContext = record
// the currently running service factory
/// it can be used within server-side implementation to retrieve the
// associated TSQLRestServer instance
/// note that TServiceFactoryServer.Get() won’t override this value, when
// called within another service (i.e. if Factory is not nil)
Factory: TServiceFactoryServer;
// the currently running context which launched the method
/// Low-Level RESTful context is also available in its Call member
/// Request.Server is the safe access point to the underlying TSQLRestServer,
/// unless the service is implemented via TInjectableObjectRest, so the
/// TInjectableObjectRest.Server property is preferred
/// make available e.g. current session or authentication parameters
/// (including e.g. user details via Request.Server.SessionGetUser)
Request: TSQLRestServerURIContext;
end;
When used, a local copy or a PServiceRunningContext pointer should better be created, since accessing a threadvar has a non negligible performance cost.

If your code is compiled within some packages, threadvar read won't work, due to a Delphi compiler/RTL restriction (bug?). In such case, you have to call the following function instead of directly access the threadvar:

```delphi
function CurrentServiceContext: TServiceRunningContext;
```

Note that this global threadvar is reset to 0 outside an interface-based service method call. It will therefore be useless to read it from a method-based service, for instance.

### 16.4.4.2. Implement your service from TInjectableObjectRest

An issue with the ServiceContext threadvar is that the execution context won't be filled when a SOA method is executed outside a client/server context, e.g. if the TSQLRestServer instance did resolve itself its dependencies using Services.Resolve().

A safer (and slightly faster) alternative is to implement your service by inheriting from the TInjectableObjectRest class. This class has its own Resolve() overloaded methods (inherited from TInjectableObject), but also two additional properties:

```delphi
TInjectableObjectRest = class(TInjectableObject)
  ...
  public
  property Factory: TServiceFactoryServer read fFactory;
  property Server: TSQLRestServer read fServer;
end;
```

Those properties will be injected by TServiceFactoryServer.CreateInstance, i.e. when the service implementation object will be instantiated on the server side. They will give direct and safe access to the underlying REST server, e.g. all its ORM methods.

### 16.4.5. Using services on the Server side

Once the service is registered on the server side, it is very easy to use it in your code.

In a complex Service-Oriented Architecture (SOA) (page 90), it is not a good practice to have services calling each other. Code decoupling is a key to maintainability here. But in some cases, you'll have to consume services on the server side, especially if your software architecture has several layers (like in a Domain-Driven Design (page 99)): your application services could be decoupled, but the Domain-Driven services (those implementing the business model) could be on another Client-Server level, with a dedicated protocol, and could have nested calls.

In this case, according to the SOLID design principles (page 389), you'd better rely on abstraction in your code, i.e. not call the service implementation (i.e. the TInterfacedObject instances or even worse directly the low-level classes or functions), but the service abstract interface. You can use the following method of your TSQLRest.Services instance (note that this method is available on both client and server sides as abstract TServiceFactory, so is the right access point to all services):

```delphi
function TServiceFactory.Get(out Obj): Boolean;
```

You have several methods to retrieve a TServiceFactory instance, either from the service name, its
GUID, or its index in the list.

That is, you may code:

```pascal
var I: ICalculator;
begin
  if ServiceContext.Request.Server.Services['Calculator'].Get(I) then
    result := I.Add(10,20);
end;
```

or, for a more complex service:

```pascal
var CN: IComplexNumber;
begin
  if not ServiceContext.Request.Server.Services.Resolve(IComplexNumber,CN) then
    exit; // IComplexNumber interface not found
  CN.Real := 0.01;
  CN.Imaginary := 3.1415;
  CN.Add(100,200);
  assert(SameValue(CN.Real,100.01));
  assert(SameValue(CN.Imaginary,203.1415));
end; // here CN will be released
```

For newer generic-aware versions of Delphi (i.e. Delphi 2010 and up, since Delphi 2009 is buggy about generics), you can use such a method, which enables compile-time checking:

```pascal
var I: ICalculator;
begin
  I := Server.Service<ICalculator>;
  if I<>nil then
    result := I.Add(10,20);
end;
```

You can of course cache/store your TServiceFactory or TSQLRest instances within a local field, if you wish. Using ServiceContext.Request.Server is verbose and error-prone. But you may consider instead to implement your service from TInjectableObjectRest (page 433): the TInjectableObjectRest class has already its built-in Resolve() overloaded methods, and direct access to the underlying Server: TSQLRestServer instance. So you will be able to write directly both SOA and ORM code:

```pascal
var I: ICalculator;
begin
  if Resolve(ICalculator,I) then
    Server.Add(TSQLRecordExecution,['Add',I.Add(10,20)]);
end;
```

If the service has been defined as sicPerThread, the instance you will retrieve on the server side will also be specific to the running thread - in this case, caching the instance may be source of confusion, since there will be one dedicated instance per thread.
16.5. Client side

There is no implementation at all on the client side. This is the magic of mORMot's services: no Wizard to call (as in DataSnap, RemObjects or WCF), nor client-side methods to write - as with our Client-Server services via methods (page 373).

You just register the existing interface definition (e.g. our ICalculator type), and you can remotely access to all its methods, executed on the server side.

In fact, a hidden "fake" TInterfaceObject class will be created by the framework (including its internal VTable and low-level assembler code), and used to interact with the remote server. But you do not have to worry about this process: it is transparent to your code.

16.5.1. Set up the Client factory

On the client side, you have to register the corresponding interface to initialize its associated factory, as such:

```pascal
Client.ServiceRegister([TypeInfo(ICalculator)],sicShared);
```

You may prefer Direct use of interface types without TypeInfo() (page 410), if the type has previously been registered:

```pascal
Client.ServiceDefine([ICalculator],sicShared);
```

It is very close to the Server-side registration, despite the fact that we do not provide any implementation class here. Implementation will remain on the server side.

Note that the implementation mode (here sicShared) shall match the one used on the server side. An error will occur if this setting is not coherent.

The other interface we talked about, i.e. IComplexNumber, is registered as such for the client:

```pascal
Client.ServiceDefine([IComplexNumber],sicClientDriven);
```

This will create the corresponding TServiceFactoryClient instance, ready to serve fake implementation classes to the client process.

To be more precise, this registration step is indeed not mandatory on the client side. If you use the TServiceContainerClient.Info() method, the client-side implementation will auto-register the supplied interface, in sicClientDriven implementation mode.

16.5.2. Using services on the Client side

Once the service is registered on the client side, it is very easy to use it in your code.

You can use the same methods as on the server side to retrieve a TServiceFactory instance.

That is, you may code:

```pascal
var I: ICalculator;
begin
  if Client.Services['Calculator'].Get(I) then
    result := I.Add(10,20);
end;
```

For Delphi 2010 and up, you can use a generic-based method, which enables compile-time checking:

```pascal
var I: ICalculator;
begin
```

I := Client.Service<ICalculator>;
if I<>nil then
  result := I.Add(10,20);
end;

For a more complex service, initialized as sicClientDriven:

var CN: IComplexNumber;
begin
  if not Client.Services.Resolve(IComplexNumber,CN) then
    exit; // IComplexNumber interface not found
  CN.Real := 0.01;
  CN.Imaginary := 3.1415;
  CN.Add(100,200);
  assert(SameValue(CN.Real,100.01));
  assert(SameValue(CN.Imaginary,203.1415));
end; // here CN will be released on both client AND SERVER sides

The code is just the same as on the server. The only functional change is that the execution will take place on the server side (using the registered TServiceComplexNumber implementation class), and the corresponding class instance will remain active until the CN local interface will be released on the client.

You can of course cache your TServiceFactory instance within a local field, if you wish. On the client side, even if the service has been defined as sicPerThread, you can safely cache and reuse the same instance, since the per-thread process will take place on the server side only.

As we stated in the previous paragraph, since the IComplexNumber is to be executed as sicClientDriven, it is not mandatory to call the Client.ServiceRegister or ServiceDefine method for this interface. In fact, during Client.Services.Info(TypeInfo(IComplexNumber)) method execution, the registration will take place, if it has not been done explicitly before. For code readability, it may be a good idea to explicitly register the interface on the client side also, just to emphasize that this interface is about to be used, and in which mode.
16.6. Sample code

You can find in the "SQLite3/Samples/14 - Interface based services" folder of the supplied source code distribution, a dedicated sample about this feature.

Purpose of this code is to show how to create a client-server service, using interfaces, over named pipe communication.

16.6.1. The shared contract

First, you'll find a common unit, shared by both client and server applications:

```pascal
unit Project14Interface;

interface

type
  ICalculator = interface(IInvokable)
  ['{9A68C8ED-CEB2-4E09-87D4-4A16F496E5FE}']
  function Add(n1,n2: integer): integer;
end;

const
  ROOT_NAME = 'service';
  PORT_NAME = '888';
  APPLICATION_NAME = 'RestService';

implementation

uses mORMot;

initialization
  TInterfaceFactory.RegisterInterfaces([TypeInfo(ICalculator)]);
end.
```

Unique purpose of this unit is to define the service interface, and the ROOT_NAME used for the ORM Model (and therefore RESTful URI scheme), and the APPLICATION_NAME used for named-pipe communication.

This ICalculator type is also registered for the internal interface factory system, so that you could use the framework methods directly with ICalculator instead of TypeInfo(ICalculator).

16.6.2. The server sample application

The server is implemented as such:

```pascal
program Project14Server;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  mORMot,
  mORMotSQLite3,
  Project14Interface;

type
  TServiceCalculator = class(TInterfacedObject, ICalculator)
  public
```

...
function Add(n1, n2: integer): integer;
end;

function TServiceCalculator.Add(n1, n2: integer): integer;
begin
  result := n1 + n2;
end;

var
  aModel: TSQLModel;
begin
  aModel := TSQLModel.Create([], ROOT_NAME);
  try
    with TSQLRestServerDB.Create(aModel, ChangeFileExt(paramstr(0), '.db'), true) do
    try
      CreateMissingTables; // we need AuthGroup and AuthUser tables
      ServiceDefine(TServiceCalculator, [ICalculator], sicShared);
      if ExportServerNamedPipe(APPLICATION_NAME) then
        writeln('Background server is running.'#10)
      else
        writeln('Error launching the server.'#10);
    finally
      Free;
    end;
  finally
    aModel.Free;
  end;
end.

It will instantiate a TSQLRestServerDB class, containing a SQLite3 database engine. In fact, since we need authentication, both AuthGroup and AuthUser tables are expected to be available.

Then a call to ServiceDefine() will define the ICalculator contract, and the TServiceCalculator class to be used as its implementation. The sicShared mode is used, since the same implementation class can be shared during all calls (there is no shared nor private data to take care).

Note that since the database expectations of this server are basic (only CRUD commands are needed to handle authentication tables), we may use a TSQLRestServerFullMemory class instead of TSQLRestServerDB. This is what is the purpose of the Project14ServerInMemory.dpr sample:

program Project14ServerInMemory;
(...)
  with TSQLRestServerFullMemory.Create(aModel, 'test.json', false, true) do
  try
    ServiceDefine(TServiceCalculator, [ICalculator], sicShared);
    if ExportServerNamedPipe(APPLICATION_NAME) then
      (...)
  finally
  end;
(...)

Using this class will include the CreateMissingTables call to create both AuthGroup and AuthUser tables needed for authentication. But the resulting executable will be lighter: only 200 KB when compiled with Delphi 7 and our LVCL classes, for a full service provider.

16.6.3. The client sample application

The client is just a simple form with two TEdit fields (edtA and edtB), and a "Call" button, which OnClick event is implemented as:

procedure TForm1.btnCallClick(Sender: TObject);
var
  a, b: integer;
  err: integer;
  I: ICalculator;
begin
  val(edtA.Text, a, err);
  I := TServiceCalculator.Create;
  I.Add(a, b);
  writeln('Result is: ' + IntToStr(I.Result));
end;
The client code is initialized as such:

- A TSQLRestClientURINamedPipe instance is created, with an associate TSQLModel and the given APPLICATION_NAME to access the proper server via a named pipe communication;
- The connection is authenticated with the default 'User' rights;
- The ICalculator interface is defined in the client's internal factory, in sicShared mode (just as in the server).

Once the client is up and ready, the local I: ICalculator variable instance is retrieved, and the remote service is called directly via a simple I.Add(a,b) statement.

You can imagine how easy and safe it will be to implement a Service-Oriented Architecture (SOA) (page 90) for your future applications, using mORMot.

### 16.6.4. Enhanced sample: remote SQL access

You will find in the SQLite3\Samples\16 - Execute SQL via services folder of mORMot source code a Client-Server sample able to access any external database via JSON and HTTP. It is a good demonstration of how to use a non-trivial interface-based service between a client and a server. It will also show how our SynDB.pas classes have a quite abstract design, and are easy to work with, whatever database provider you need to use.

The corresponding service contract has been defined:

```
TRemoteSQLEngine = (rseOleDB, rseODBC, rseOracle, rseSQLite3, rseJet, rseMSSQL);

IRemoteSQL = interface(IInvokable)

    ['{9A6BC8ED-CEB2-4E09-87D4-4A16F496E5FE}']
    procedure Connect(aEngine: TRemoteSQLEngine; const aServerName, aDatabaseName,
                        aUserID, aPassWord: RawUTF8);
    function GetTableNames: TRawUTF8DynArray;
    function Execute(const aSQL: RawUTF8; aExpectResults, aExpanded: Boolean): RawJSON;
end;
```

Purpose of this service is:

- To Connect() to any external database, given the parameters as expected by a standard TSQLDBConnectionProperties.Create() constructor call;
- To retrieve all table names of this external database as a list;
- To execute any SQL statement, returning the content as JSON array, ready to be consumed by AJAX applications (if aExpanded is true), or a Delphi client (e.g. via a TSQLTableJSON and the mORMotUI unit).
Of course, this service will be defined in SICClientDriven mode. That is, the framework will be able to manage a client-driven TSQLDBProperties instance life time.

Benefit of this service is that no database connection is required on the client side: a regular HTTP connection is enough. No need to neither install nor configure any database provider.

Due to mORMot optimized JSON serialization, it will probably be faster to work with such plain HTTP / JSON services, instead of a database connection through a VPN. In fact, database connections are made to work on a local network, and do not like high-latency connections, which are typical on the Internet. On the contrary, the mORMot Client-Server process is optimized for such kind of connection.

Note that the Execute() method returns a RawJSON kind of variable, which is in fact a sub-type of RawUTF8. Its purpose is to transmit the UTF-8 encoded content directly, with no translation to a JSON string, as will be the case with a RawUTF8 variable. In fact, escaping some JSON array within a JSON string is quite verbose. Using RawJSON in this case ensure the best client-side and server-side speed, and also reduce the transmission bandwidth.

The server part is quite easy to follow:

type
  TServiceRemoteSQL = class(TInterfacedObject, IRemoteSQL)
    protected
      fProps: TSQLDBConnectionProperties;
    public
      destructor Destroy; override;
    public // implements IRemoteSQL methods
      procedure Connect(aEngine: TRemoteSQLEngine; const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8);
      function GetTableNames: TRawUTF8DynArray;
      function Execute(const aSQL: RawUTF8; aExpectResults, aExpanded: Boolean): RawJSON;
    end;

{ TServiceRemoteSQL }

procedure TServiceRemoteSQL.Connect(aEngine: TRemoteSQLEngine; const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8);
const // rseOleDB, rseODBC, rseOracle, rseSQLite3, rseJet, rseMSSQL
  TYPES: array[TRemoteSQLEngine] of TSQLDBConnectionPropertiesClass = (TOleDBConnectionProperties, TODBCConnectionProperties, TSQLDBOracleConnectionProperties, TSQLDBSQLite3ConnectionProperties, TOleDBJetConnectionProperties, TOleDBMSSQL2008ConnectionProperties);
begin
  if fProps<>nil then
    raise Exception.Create('Connect called more than once');
  fProps := TYPES[aEngine].Create(aServerName,aDatabaseName,aUserID,aPassWord);
end;

function TServiceRemoteSQL.Execute(const aSQL: RawUTF8; aExpectResults, aExpanded: Boolean): RawJSON;
var
  res: ISQLDBRows;
begin
  if fProps=nil then
    raise Exception.Create('Connect call required before Execute');
  res := fProps.ExecuteInlined(aSQL,aExpectResults);
  if res=nil then
    result := '' else
    result := res.FetchAllAsJSON(aExpanded);
end;

function TServiceRemoteSQL.GetTableNames: TRawUTF8DynArray;
begin
  if fProps=nil then
    raise Exception.Create('Connect call required before GetTableNames');
fProps.GetTableNames(result);
end;

destructor TServiceRemoteSQL.Destroy;
begin
  FreeAndNil(fProps);
  inherited;
end;

Any exception during SynDB.pas process, or raised manually in case of wrong use case will be transmitted to the client, just as expected. The fProps instance life-time is handled by the client, so all we need is to release its pointer in the service implementation destructor.

The services are initialized on the server side with the following code:

```pascal
var
  aModel: TSQLModel;
  aServer: TSQLRestServer;
  aHTTPServer: TSQLHttpServer;
begin
  // define the log level
  with TSQLLog.Family do begin
    Level := LOG_VERBOSE;
    EchoToConsole := LOG_VERBOSE; // log all events to the console
  end;
  // manual switch to console mode
  AllocConsole;
  TextColor(ccLightGray);
  // create a Data Model
  aModel := TSQLModel.Create([],ROOT_NAME);
  try
    // initialize a TObjectList-based database engine
    aServer := TSQLRestServerFullMemory.Create(aModel,'users.json',false,true);
    try
      // register our IRemoteSQL service on the server side
      aServer.ServiceRegister(TServiceRemoteSQL,[TypeInfo(IRemoteSQL)],sicClientDriven).
      // fProps should better be executed and released in the one main thread
      SetOptions([],[optExecInMainThread,optFreeInMainThread]);
      // Launch the HTTP server
      aHTTPServer := TSQLHttpServer.Create('888',[aServer],'+',useHttpApiRegisteringURI);
      try
        aHTTPServer.AccessControlAllowOrigin := '*'; // for AJAX requests to work
        writeln(#10'Background server is running.'#10);
        writeln('Press [Enter] to close the server.'#10);
        ConsoleWaitForEnterKey;
        finally
          aHTTPServer.Free;
        end;
      finally
        aServer.Free;
      end;
    finally
      aModel.Free;
    end;
  finally
    aServer.Free;
  end;
end.
```

This is a typical mORMot server initialization, published over the HTTP communication protocol (with auto-registration feature, if possible, as stated by the useHttpApiRegisteringURI flag). Since we won’t use ORM for any purpose but authentication, a fast TObjectList-based engine (i.e. TSQLRestServerFullMemory) is enough for this sample purpose.

In the above code, you can note that IRemoteSQL service is defined with the optExecInMainThread and optFreeInMainThread options. It means that all methods will be executed in the main process thread. In practice, since SynDB.pas database access may open one connection per thread (e.g. for OleDb / MS SQL or Oracle providers), it may use a lot of memory. Forcing the database execution in
the main thread will lower the resource consumption, and still will perform with decent speed (since all the internal marshalling and communication will be multi-threaded in the framework units).

From the client point of view, it will be consumed as such:

```pascal
procedure TFormMain.FormShow(Sender: TObject);
begin
  (...)  
  fModel := TSQLModel.Create([], ROOT_NAME);
  fClient := TSQLHttpClient.Create('localhost', '8888', fModel);
  if not fClient.ServerTimestampSynchronize then begin
    ShowLastClientError(fClient, 'Please run Project16ServerHttp.exe');
    Close;
    exit;
  end;
  if (not fClient.SetUser('User', 'synopse')) or
     (not fClient.ServiceRegisterClientDriven(TypeInfo(IRemoteSQL), fService)) then begin
    ShowLastClientError(fClient, 'Remote service not available on server');
    Close;
    exit;
  end;
end;
```

Our IRemoteSQL service will be accessed in sIClientDriven mode, so here we need to initialize RESTful authentication - see below (page 546) - with a proper call to SetUser().

Note the use of ShowLastClientError() function of mORMotUILogin unit, which is able to use our SynTaskDialog unit to report standard and detailed information about the latest error.

In this sample, no table has been defined within the ORM model. It is not necessary, since all external process will take place at the SQL level. As we need authentication (see the call to fClient.SetUser method), the ORM core will by itself add the TSQLAuthUser and TSQLAuthGroup tables to the model - no need to add them explicitly.

From now on, we have a fService: IRemoteSQL instance available to connect and process any remote SQL request.

```pascal
procedure TFormMain.btnOpenClick(Sender: TObject);
var TableNames: TRawUTF8DynArray;
begin
  (...)  
  with fSettings do
  begin
    fService.Connect(Engine, ServerName, DatabaseName, UserID, PassWord);
    TableNames := fService.GetTableNames;
    cbbTableNames.Items.Text := UTF8ToString(RawUTF8ArrayToCSV(TableNames, #13#10));
  end;
end;
```

Now we are connected to the database via the remote service, and we retrieved the table names in a TComboBox.

Then a particular SQL statement can be executed as such:

```pascal
procedure TFormMain.btnExecuteClick(Sender: TObject);
var SQL: RawUTF8;
begin
  SQL := trim(StringToUTF8(mmoQuery.Text));
  Screen.Cursor := crHourGlass;
  try
    try
      if isSelect(pointer(SQL)) then begin
        fTableJSON := fService.Execute(SQL, True, False);
        TSQLTableToGrid.Create(drwgrdData,
          TSQLTableJSON.Create([], SQL, pointer(fTableJSON), Length(fTableJSON)), fClient);
      end else
      fService.Execute(SQL, False, False);
    except
      on E: Exception do
      end;
  end;
end;
```
Here, TSQLTableToGrid.Create(), from the mORMotUI unit, will "inject" the returned data to a standard TDrawGrid, using a TSQLTableJSON instance to un-serialize the returned JSON content.

Note that in case of any exception (connection failure, or server side error, e.g. wrong SQL statement), the ShowException() method is used to notify the user with appropriate information.
16.7. Asynchronous callbacks

When publishing SOA services, most of them are defined as stateless, in a typical query/answer pattern - see Service-Oriented Architecture (SOA) (page 90). This fits exactly with the RESTful approach of Client-Server services via interfaces (page 419), as proposed by the framework.

But it may happen that a client application (or service) needs to know the state of a given service. In a pure stateless implementation, it will have to query the server for any state change, i.e. for any pending notification - this is called polling.

Polling may take place for instance:
- When a time consuming work is to be processed on the server side. In this case, the client could not wait for it to be finished, without raising a timeout on the HTTP connection: as a workaround, the client may start the work, then ask for its progress status regularly using a timer and a dedicated method call;
- When an unpredictable event is to be notified from the server side. In this case, the client should ask regularly (using a timer, e.g. every second), for any pending event, then react on purpose.

It may therefore sounds preferred, and in some case necessary, to have the ability to let the server notify one or several clients without any prior query, nor having the requirement of a client-side timer:
- Polling may be pretty resource consuming on both client and server sides, and add some unwanted latency;
- If immediate notification is needed, some kind of "long polling" algorithm may take place, i.e. the server will wait for a long time before returning the notification state if no event did happen: in this case, a dedicated connection is required, in addition to the REST one;
- In an event-driven systems, a lot of messages are sent to the clients: a proper publish/subscribe mechanism is preferred, otherwise the complexity of polling methods may increase and become inefficient and unmaintainable;
- Explicit push notifications may be necessary, e.g. when a lot of potential events, associated with a complex set of parameters, are likely to be sent by the client.

Our mORMot framework is therefore able to easily implement asynchronous callbacks over WebSockets, defining the callbacks as interface parameters in service method definitions - see Service Methods Parameters (page 423).

16.7.1. WebSockets support

By definition, HTTP connections are stateless and one-way, i.e. a client sends a request to the server, which replies back with an answer. There is no way to let the server send a message to the client, without a prior request from the client side.

WebSockets is a communication protocol which is able to upgrade a regular HTTP connection into a dual-way communication wire. After a safe handshake, the underlying TCP/IP socket is able to be accessed directly, via a set of lightweight frames over an application-defined protocol, without the HTTP overhead.

The SynBidirSock.pas unit implements low-level server and client WebSockets communication.

The TWebSocketProtocol class defines an abstract WebSockets protocol, currently implemented as several classes:
For our *Client-Server services via interfaces* (page 419), we will still need to make *RESTful* requests, so the basic *WebSockets* framing has been enhanced to support TWebSocketProtocolRest REST-compatible protocols, able to use the single connection for both REST queries and asynchronous notifications.

Two classes are available for your SOA applications:

- TWebSocketProtocolJSON as a "pure" JSON light protocol;
- TWebSocketProtocolBinary as a binary proprietary protocol, with optional frame compression and AES encryption (using AES-NI hardware instructions, if available).

In practice, on the server side, you will start your TSQLHttpServer by specifying useBidirSocket as kind of server:

```delphi
HttpServer := TSQLHttpServer.Create('8888',[Server],'+',useBidirSocket);
```

Under the hood, it will instantiate a TWebSocketServer HTTP server, as defined in mORMotHttpServer.pas, based on the sockets API, able to upgrade the HTTP protocol into *WebSockets*. Our *High-performance http.sys server* (page 326) is not yet able to switch to *WebSockets* - and at API level, it will require at least *Windows 8* or *Windows 2012 Server*.

Then you enable *WebSockets* for the TWebSocketProtocolBinary protocol, with a symmetric encryption key:

```delphi
HttpServer.WebSocketsEnable(Server,'encryptionkey');
```

On the client side, you will use a TSQLHttpclientWebsockets instance, as defined in mORMotHttpclient.pas, then explicitly upgrade the connection to use *WebSockets* (since by default, it will stick to the HTTP protocol):

```delphi
Client := TSQLHttpclientwebsockets.Create('127.0.0.1','8888',TSQLModel.Create([]));
Client.WebSocketsUpgrade('encryptionkey');
```

The expected protocol detail should match the one on the server, i.e. 'encryptionkey' encryption over our binary protocol.

Once upgraded to *WebSockets*, you may use regular REST commands, as usual:

```delphi
Client.ServerTimestampSynchronize;
```

But in addition to regular query/answer commands as defined for *Client-Server services via interfaces* (page 419), you will be able to define callbacks using interface parameters to the service methods.

Under the hood, both client and server will communicate using *WebSockets* frames, maintaining the connection active using heartbeats (via ping/pong frames), and with clean connection shutdown, from any side. You can use the Settings property of the TWebSocketServerRest instance, as returned by TSQLHttpServer.WebSocketsEnable(), to customize the low-level *WebSockets* protocol (e.g. timeouts or heartbeats) on the server side. The TSQLHttpclientWebsockets.WebSockets.Settings
property will allow the same, on the client side.

We have observed, from our regression tests and internal benchmarking, that using our WebSockets may be faster than regular HTTP, since its frames will be sent as once, whereas HTTP headers and body are not sent in the same TCP packet, and compression will be available for the whole frame, whereas HTTP headers are not compressed. The ability to use strong AES encryption will make this mean of communication even safer than plain HTTP, even with AES encryption over HTTP (page 333).

16.7.1.1. Using a "Saga" callback to notify long term end-of-process

An example is better than 100 talks. So let's take a look at the Project31LongWorkServer.dpr and Project31LongWorkClient.dpr samples, from the SQLite3\Samples\31 - WebSockets sub-folder. They will implement a client/server application, in which the client launches a long term process on the server side, then is notified when the process is done, either with success, or failure. Such a pattern is very common in the SOA world, and also known as "saga" - see http://www.rgoarchitects.com/Files/SOAPatterns/Saga.pdf - but in practice, it may be difficult to implement it safely and easily. Let's see how our framework make writing sagas a breeze.

First we define the interfaces to be used, in a shared Project31LongWorkCallbackInterface.pas unit:

type
   ILongWorkCallback = interface(IInvokable)
      ['{425BF199-19C7-4B2B-B1A4-A58E7A9A4748}']
      procedure WorkFinished(const workName: string; timeTaken: integer);
      procedure WorkFailed(const workName, error: string);
   end;

   ILongWorkService = interface(IInvokable)
      ['{09FDFCEF-86E5-4077-80D8-6618801A9224A}']
      procedure StartWork(const workName: string; const onFinish: ILongWorkCallback);
      function TotalWorkCount: Integer;
   end;

The only specific definition is the const onFinish: ILongWorkCallback parameter, supplied to the ILongWorkService.StartWork() method. The client will create a class implementing ILongWorkCallback, then specify it as parameter to this method. On the server side, a "fake" class will implement ILongWorkCallback, then will call back the client using the very same WebSockets connection, when any of its methods will be executed.

As you can see, a single callback interface instance may have several methods, with their own set of parameters (here WorkFinished and WorkFailed), so that the callback may be quite expressive. Any kind of usual parameters will be transmitted, after serialization: string, integer, but even record, dynamic arrays, TSQLRecord or TPersistent values.

When the ILongWorkCallback instance will be released on the client side, the server will be notified, so that any further notification won't create a connection error. We will see later how to handle those events.

16.7.1.2. Client service consumption

The client may be connected to the server as such (see the Project31LongWorkClient.dpr sample source code for the full details, including error handling):

```-delphi
var
  Client: TSQLHttpClientWebsockets;
  workName: string;
```

```delphi```

As you can see, a single callback interface instance may have several methods, with their own set of parameters (here WorkFinished and WorkFailed), so that the callback may be quite expressive. Any kind of usual parameters will be transmitted, after serialization: string, integer, but even record, dynamic arrays, TSQLRecord or TPersistent values.

When the ILongWorkCallback instance will be released on the client side, the server will be notified, so that any further notification won't create a connection error. We will see later how to handle those events.
Service: ILongWorkService;
callback: ILongWorkCallback;
begin
  Client := TSQLHttpClientWebsockets.Create('127.0.0.1','8888',TSQLModel.Create([]));
  Client.WebSocketsUpgrade(PROJECT31_TRANSITION_KEY);
  Client.ServiceDefine([ILongWorkService],sicShared);
  Client.Services.Resolve(ILongWorkService,Service);

Then we define our callback, using a dedicated class:

type
  TLongWorkCallback = class(TInterfacedCallback,ILongWorkCallback)
protected
  procedure WorkFinished(const workName: string; timeTaken: integer);
  procedure WorkFailed(const workName, error: string);
end;

procedure TLongWorkCallback.WorkFailed(const workName, error: string);
begin
  writeln(#13'Received callback WorkFailed(',workName,',') with message ",',error,"');
end;

procedure TLongWorkCallback.WorkFinished(const workName: string;
  timeTaken: integer);
begin
  writeln(#13'Received callback WorkFinished(',workName,',') in ',timeTaken,'ms');
end;

Then we specify this kind of callback as parameter to start a long term work:

callback := TLongWorkCallback.Create(Client,ILongWorkCallback);
try
  repeat
    readln(workName);
    if workName='' then
      break;
    Service.StartWork(workName,callback);
  until False;
finally
  callback := nil; // the server will be notified and release its "fake" class
  Service := nil; // release the service local instance BEFORE Client.Free
end;

As you can see, the client is able to start one or several work processes, then expects to be notified of the process ending on its callback interface instance, without explicitly polling the server for its state, since the connection was upgraded to WebSockets via a call to TSQLHttpClientWebsockets.WebSocketsUpgrade().

16.7.1.3. Server side implementation

The server will define the working thread as such (see the Project31LongWorkServer.dpr sample source code for the full details):

type
  TLongWorkServiceThread = class(TThread)
protected
  fCallback: ILongWorkCallback;
  fWorkName: string;
  procedure Execute; override;
public
  constructor Create(const workName: string; const callback: ILongWorkCallback);
end;

constructor TLongWorkServiceThread.Create(const workName: string;
  const callback: ILongWorkCallback);
begin
inherited Create(false);
  fCallback := Callback;
  fWorkName := workName;
  FreeOnTerminate := true;
end;

procedure TLongWorkServiceThread.Execute;
var
  tix: Int64;
begin
  tix := GetTickCount64;
  Sleep(5000+Random(1000)); // some hard work
  if Random(100)>20 then
    fCallback.WorkFinished(fWorkName,GetTickCount64-tix) else
    fCallback.WorkFailed(fWorkName,'expected random failure');
end;

The callback is expected to be supplied as a ILongWorkCallback interface instance, then stored in a fCallback protected field for further notification.

Some work is done in the TLongWorkServiceThread.Execute method (here just a Sleep() of more than 5 seconds), and the end-of-work notification is processed, as success or failure (depending on random in this fake process class), on either of the ILongWorkCallback interface methods.

The following class will define, implement and register the ILongWorkService service on the server side:

type
  TLongWorkService = class(TInterfacedObject,ILongWorkService)
    protected
      fTotalWorkCount: Integer;
    public
      procedure StartWork(const workName: string; const onFinish: ILongWorkCallback);
      function TotalWorkCount: Integer;
    end;

procedure TLongWorkService.StartWork(const workName: string; const onFinish: ILongWorkCallback);
begin
  InterlockedIncrement(fTotalWorkCount);
  TLongWorkServiceThread.Create(workName,onFinish);
end;

function TLongWorkService>TotalWorkCount: Integer;
begin
  result := fTotalWorkCount;
end;

var
  HttpServer: TSQLHttpServer;
  Server: TSQLRestServerFullMemory;
begin
  Server := TSQLRestServerFullMemory.CreateWithOwnModel([]);
  Server.ServiceDefine(TLongWorkService,ILongWorkService,sicShared);
  HttpServer := TSQLHttpServer.Create('8888',[Server],'+',useBidirSocket);
  HttpServer.WebSocketsEnable(Server,PROJECT31_TRANSMISSION_KEY);
 ...

Purpose of those methods is just to create and launch the TLongWorkServiceThread process from a client request, then maintain a total count of started works, in a sicShared service instance - see "Instances life time implementation" (page 429) - hosted in a useBidirSocket kind of HTTP server.

We have to explicitly call TSQLHttpServer.WebSocketsEnable() so that this server will be able to upgrade to our WebSockets protocol, using our binary framing, and the very same symmetric encryption key as on the client side - shared as a PROJECT31_TRANSMISSION_KEY constant in the sample, but which may be safely stored on both sides.
16.7.2. Publish-subscribe for events

In event-driven architectures, the \textit{publish-subscribe} messaging pattern is a way of letting senders (called \textit{publishers}) transmit messages to their receivers (called \textit{subscribers}), without any prior knowledge of who those subscribers are. In practice, the \textit{subscribers} will express interest for a set of messages, which will be sent by the \textit{publisher} to all the \textit{subscribers} of a given message, as soon as it is be notified.

\textbf{Publish-Subscribe Pattern}

In our \textit{Client-Server services via interfaces} (page 419) implementation, messages are gathered in \textit{interface} types, and each message defined as a single method, their content being the methods parameters.

Most of the SOA alternative (in Java or C#) do require class definition for messages. Our KISS approach will just use method parameters values as message definition.

To maintain a list of \textit{subscribers}, the easiest is to store a \textit{dynamic array} of \textit{interface} instances, on the \textit{publisher} side.

16.7.2.1. Defining the interfaces

We will now implement a simple \textit{chat} service, able to let several clients communicate together, broadcasting any message to all the other connected instances. This sample is also located in the the SQLite3\Samples\31 - WebSockets sub-folder, as Project31ChatServer.dpr and Project31ChatClient.dpr.

So you first define the callback interface, and the service interface:

\begin{verbatim}
type
  IChatCallback = interface(IInvokable)
    ['{EA7EE51-3EBA-4B47-A356-25337451BD1D}']
    procedure NotifyBlaBla(const pseudo, msg: string);
  end;

  IChatService = interface(IInvokable)
    ['{C92DCBEA-C680-40BD-8D9C-3E6F2ED9C9CF}']
    procedure Join(const pseudo: string; const callback: IChatCallback);
    procedure BlaBla(const pseudo,msg: string);
    procedure CallbackReleased(const callback: IInvokable; const interfaceName: RawUTF8); 
  end;
\end{verbatim}

The main command of the IChatService service is BlaBla(), which should be propagated to all client instance having joined the conversation, via IChatCallback.NotifyBlaBla() events.

Those interface types will be shared by both server and client sides, in the common Project31ChatCallbackInterface.pas unit. The definition is pretty close to what we wrote when \textit{Using a "Saga" callback to notify long term end-of-process} (page 446). For instance, if 3 people did join the chat room, the following process should take place:
The only additional method is `IChatServer.CallbackReleased()`, which, by convention, will be called on the server side when any callback interface instance is released on the client side.

As such, the `IChatService.Join()` method will implement the *subscription* to the chat service, whereas `IChatServer.CallbackReleased()` will be called when the client-side callback instance will be released (i.e. when its variable will be assigned to nil), to *unsubscribe* for the chat service.

### 16.7.2.2. Writing the Publisher

On the server side, each call to `IChatService.Join()` will *subscribe* to an internal list of connections, simply stored as an array of `IChatCallback`:

```pascal
type
  TChatService = class(TInterfacedObject, IChatService)
  protected
    fConnected: array of IChatCallback;
  public
    procedure Join(const pseudo: string; const callback: IChatCallback);
    procedure BlaBla(const pseudo, msg: string);
    procedure CallbackReleased(const callback: IInvokable; const interfaceName: RawUTF8);
  end;

procedure TChatService.Join(const pseudo: string;
  const callback: IChatCallback);
begin
  InterfaceArrayAdd(fConnected, callback);
end;
```

The `InterfaceArrayAdd()` function, as defined in `SynCommons.pas`, is a simple wrapper around any *dynamic array* of interface instances, so that you may use it, or the associated `InterfaceArrayFind()` or `InterfaceArrayDelete()` functions, to maintain the list of subscriptions.

Then a remote call to the `IChatService.BlaBla()` method should be broadcasted to all connected clients, just by calling the `IChatCallback.BlaBla()` method:

```pascal
procedure TChatService.BlaBla(const pseudo, msg: string);
var i: integer;
begin
  for i := 0 to high(fConnected) do
    fConnected[i].NotifyBlaBla(pseudo, msg);
end;
```

Note that every call to `IChatCallback.BlaBla()` within the loop will be made via *WebSockets*, in an asynchronous and non blocking way, so that even in case of huge number of clients, the `IChatService.BlaBla()` method won't block. In case of high numbers of messages, the framework is
even able to gather push notification messages into a single bigger message, to reduce the resource use - see Real-time synchronization (page 183).

If you are a bit paranoid, you may ensure that the notification process will continue, if any of the event failed:

```pascal
procedure TChatService.BlaBla(const pseudo, msg: string);
var
  i: integer;
begin
  for i := high(fConnected) downto 0 do // downwards for InterfaceArrayDelete()
  try
    fConnected[i].NotifyBlaBla(pseudo,msg);
  except
    InterfaceArrayDelete(fConnected,i); // unsubscribe the callback on failure
  end;
end;
```

This safer implementation will unregister any failing callback. If the notification raised an exception, it will ensure that this particular invalid subscriber won’t be notified any more. Note that since we may reduce the `fConnected[]` array size on the fly, the loop is processed downwards, to avoid any access violation.

On the server side, the service implementation has been registered as such:

```pascal
Server.ServiceDefine(TChatService,[IChatService],sicShared).
SetOptions([],[optExecLockedPerInterface]);
```

Here, the `optExecLockedPerInterface` option has been set, so that all method calls will be made thread-safe: concurrent access to the internal `fConnected[]` list will be protected by a lock. Since a global list of connections is to be maintained, the service life time has been defined as `sicShared` - see Instances life time implementation (page 429).

The following method will be called by the server, when a client callback instance is released (either explicitly, or if the connection is broken), so could be used to unsubscribe to the notification, simply by deleting the callback from the internal `fConnected[]` array:

```pascal
procedure TChatService.CallbackReleased(const callback: IInvokable; const interfaceName: RawUTF8);
begin
  if interfaceName='IChatCallback' then
    InterfaceArrayDelete(fConnected,callback);
end;
```

The framework will in fact recognize the following method definition in any interface type for a service (it will check the method name, and the method parameters):

```pascal
procedure CallbackReleased(const callback: IInvokable; const interfaceName: RawUTF8);
```

When a callback interface parameter (in our case, `IChatCallback`) will be released on the client side, this method will be called with the corresponding interface instance and type name as parameters. You do not have to call explicitly any method on the client side to unsubscribe a service: assigning `nil` to a callback variable, or freeing the class instance owning it as a field on the subscriber side, will automatically unregister it on the publisher side.

16.7.2.3. Consuming the service from the Subscriber side

On the client side, you implement the `IChatCallback` callback interface:

```pascal
type
  TChatCallback = class(TInterfacedCallback,IChatCallback)
    protected
      procedure NotifyBlaBla(const pseudo, msg: string);
    end;
```
procedure TChatCallback.NotifyBlaBla(const pseudo, msg: string);
begin
  writeln('#13'@',pseudo,' ',msg);
end;

The TInterFacedCallback type defines a TInterfacedObject sub-class, which will automatically notify the REST server when it is released. By providing the client TSQLRest instance to the TChatCallback.Create() constructor, you will ensure that the IChatService.CallbackReleased method will be executed on the server side, when the TChatCallback/IChatCallback instance will be released on the client side.

Then you subscribe to your remote service as such:

```pascal
var Service: IChatService;
    callback: IChatCallback;
    ...
    Client.ServiceDefine([IChatService],sicShared);
    if not Client.Services.Resolve(IChatService,Service) then
      raise EServiceException.Create('Service IChatService unavailable');
    ...
    callback := TChatCallback.Create(Client,IChatCallback);
    Service.Join(pseudo,callback);
    ... try
      repeat
        readln(msg);
        if msg='' then
          break;
        Service.BlaBla(pseudo,msg);
      until false;
    finally
      callback := nil; // will unsubscribe from the remote publisher
      Service := nil; // release the service local instance BEFORE Client.Free
    end;
```

You could easily implement more complex publish/subscribe mechanisms, including filtering, time to live or tuned broadcasting, by storing some additional information to the interface instance (e.g. some value to filter, a timestamp). A dynamic array of dedicated records - see TDynArray dynamic array wrapper (page 107), or a list of class instances, may be used to store the subscribers expectations.

### 16.7.2.4. Subscriber multiple redirection

Sometimes, in a complex business system, you will define several uncoupled parts of your code subscribing to the same service events. In a DDD architecture, it will be typically happen when several domain bounded contexts subscribe to a single event source, implemented in the infrastructure layer.

The easiest implementation path is to have each part registering from its side. But it will induce some redundant traffic with the publisher. And it will most probably end-up with duplicated code on subscribers side.

You may try TSQLRest.MultiRedirect and register once to a remote service, then use an internal registration mechanism to have every part of your business logic registering and consuming the events. The method returns an IMultiCallbackRedirect interface, allowing registration of sub-callbacks, with an optional set of method names, if only a sub-set of events are needed.

Note that sub-callbacks do not need to inherit from the TInterFacedCallback type: a regular TInterFacedObject is enough. They will be automatically unregistered from the internal list, if they raise an exception.
16.7.2.5. Proper threaded implementation

A mORMot multi-threaded server will use critical sections to protect shared data, and avoid potential race conditions. But even on client side, callbacks will be executed in the context of the WebSockets transmission thread. And in a typical micro-services or event-driven architecture, most nodes are clients and servers at the same time, creating a peer-to-peer mesh of services. So you should prevent any race conditions in each and every node, by protecting access to any shared data.

Likewise, if your callback triggers another method which shares the same critical section in another thread, you may encounter deadlock issues. If an event triggers a callback within a critical section used to protect a shared resource, and if this callback runs a blocking REST request, the REST answer will be received in the context of the transmission thread. If this answer tries to access the same shared resource, there will be a conflict with the main critical section lock, so the execution will lock.

To implement proper thread-safety of your callback process you could follow some patterns.

- Use several small critical sections, protecting any shared data, with the smallest granularity possible. You may use TSynLocker mutex or TLockedDocVariant schema-less storage.
- In your regression tests, ensure you run multi-threaded scenarios, with parallel requests. You may find in TSynParallel1e1Process an easy way of running concurrent client/server tests. It will help finding out most obvious implementation issues.
- By definition, most deadlocks are difficult to reproduce - they are some kind of "Heisenbugs". You may ensure proper logging of the callback process, so that you will be able to track for any deadlock which may occur on production.
- A good idea may be to gather all non-blocking callback process in a background thread using TSQLRest.AsynchRedirect. This method will implement any interface via a fake class, which will redirect all methods calls into calls of another interface, but as a FIFO in a background thread. So you will ensure that all callback process will take place in a single thread, avoiding most concurrency issues. As a side effect, the internal FIFO will leverage other threads, so may help scaling your system. For a client application using some User Interface, see below (page 453) a lock-free alternative.

Multi-threading is the key to performance. But it is also hard to properly implement. By following those simple rules, you may reduce the risk of concurrency issues.

16.7.2.6. Interacting with UI/VCL

As we have stated, all callback notifications do take place in the transmission thread, i.e. in the TWebSocketProcessClientThread instance corresponding to each connected client.

You may be tempted to use the VCL Synchronize() method, as usual, to forward the notifications to the UI layer. Unfortunately, this may trigger some unexpected concurrency issue, e.g. when asynchronous notifications (e.g. TChatCallback.NotifyBlaBla()) are received during a blocking REST command (e.g. Service.BlaBla()). The Synchronize call within the blocking command will avoid any incoming asynchronous notification wait for the main thread to be available, and will block the reception of the answer of the pending REST command...

If you experiment random hangouts of your User Interface, and 404 errors corresponding to a low-level WebSockets timeout, even when closing the application, you have certainly hit such a deadlock.

Get rid of all your Synchronize() calls! Use Windows messages instead: they are safe, efficient and fast. The framework allows to forward all incoming notifications as a dedicated Windows message in a single line:

```pascal
Client.ServiceNotificationMethodViaMessages(MainForm.Handle, WM_SERVICENOTIFICATION);
```
The WM_SERVICENOTIFICATION should have been defined as a custom user message:

```delphi
const
 WM_SERVICENOTIFICATION = WM_USER; // may be WM_USER+1, +2, ...
```

Then, the TFormMain should execute the message, as a regular event handler:

```delphi
 TFormMain = class(TForm)
 ...
  procedure ServiceNotification(var Msg: TMessage); message WM_SERVICENOTIFICATION;
 ...
  procedure TFormMain.ServiceNotification(var Msg: TMessage);
 begin
    TSQLRestClientURI.ServiceNotificationMethodExecute(Msg);
  end;
```

Thanks to these two lines, the callbacks will be executed asynchronously in the main UI thread, using the optimized Message queue of the Operating System, without any blocking execution, nor race condition.

### 16.7.3. Interface callbacks instead of class messages

If you compare with existing client/server SOA solutions (in Delphi, Java, C# or even in Go or other frameworks), this interface-based callback mechanism sounds pretty unique and easy to work with.

Most Events Oriented solutions do use a set of dedicated messages to propagate the events, with a centralized Message Bus (like MSMQ or JMS), or a P2P approach (see e.g. ZeroMQ or NanoMsg). In practice, you are expected to define one class per message, the class fields being the message values. You will define e.g. one class to notify a successful process, and another class to notify an error. SOA services will eventually tend to be defined by a huge number of individual classes, with the temptation of re-using existing classes in several contexts.

Our interface-based approach allows to gather all messages:
- In a single interface type per notification, i.e. probably per service operation;
- With one method per event;
- Using method parameters defining the event values.

Since asynchronous notifications are needed most of the time, method parameters will be one-way, i.e. only const. Blocking request may also be defined, as we will see below (page 456). And an evolved algorithm will transparently gather outgoing messages, to enhance scalability.

Behind the scene, the framework will still transmit raw messages over IP sockets, like other systems, but events notification will benefit from using interfaces, on both server and client sides.

#### 16.7.3.1. Using service and callback interfaces

For instance, you may define the following generic service and callback to retrieve a file from a remote camera, using mORMot's interface-based approach:

```delphi
type
 // define some custom types to make the implicit explicit
 TCameraID = RawUTF8;
 TPictureID = RawUTF8;
 // mORMot notifications using a callback interface definition
 IMyCameraCallback = interface(IInvokable)
 ['\445F\967F-\79C0-4735-A972-0BED6CC63D1D']
  procedure Started(const Camera: TCameraID; const Picture: TPictureID);
  procedure Progressed(const Camera: TCameraID; const Picture: TPictureID;
                         CurrentSize, TotalSize: cardinal);
  procedure Finished(const Camera: TCameraID; const Picture: TPictureID;
```

```delphi```
const PublicURI: RawUTF8; TotalSize: cardinal);
procedure ErrorOccurred(const Camera: TCameraID; const Picture: TPictureID;
const MessageText: RawUTF8);
end;
// mORMot main service, also defined as an interface
IMyCameraService = interface(IInvokable)
[ '{3CE61E74-A01D-41F5-A414-94F204F140E1}' ]
function TakePicture(const Camera: TCameraID; const Callback: IMyCameraCallback): TPictureID;
end;

Take a deep breath, and keep in mind those two type definitions as reference. In a single look, I guess you did get the expectation of the "Camera Service". We will now compare with a classical message-based pattern.

16.7.3.2. Classical message(s) event

With a class-based message kind of implementation, you will probably define a single class, containing all potential information:

type
// a single class message will need a status
TMyCameraCallbackState = (
  ccsStarted, ccsProgressed, ccsFinished, ccsErrorOccurred);
// the single class message
TMyCameraCallbackMessage = class
private
  fCamera: TCameraID;
  fPicture: TPictureID;
  fTotalSize: cardinal;
  fMessageText: RawUTF8;
  fState: TMyCameraCallbackState;
published
  property State: TMyCameraCallbackState read fState write fState;
  property Camera: TCameraID read fCamera write fCamera;
  property Picture: TPictureID read fPicture write fPicture;
  property TotalSize: cardinal read fTotalSize write fTotalSize;
  property MessageText: RawUTF8 read fMessageText write fMessageText;
end;

This single class is easy to write, but makes it a bit confusing to consume the notification. Which field comes with which state? The client-side code will eventually consist of a huge case mMessage.State of ... block, with potential issues. The business logic does not appear in this type definition. Easy to write, difficult to read - and maintain...

In order to have an implementation closer to SOLID design principles (page 389), you may define a set of classes, as such:

type
// all classes will inherit from this one, to have common properties
TMyCameraCallbackAbstract = class
private
  fCamera: TCameraID;
  fPicture: TPictureID;
published
  property Camera: TCameraID read fCamera write fCamera;
  property Picture: TPictureID read fPicture write fPicture;
end;
// message class when the picture acquisition starts
TMyCameraCallbackStarted = class(TMyCameraCallbackAbstract);
// message class when the picture is acquired
TMyCameraCallbackFinished = class(TMyCameraCallbackAbstract)
private
  fPublicURI: RawUTF8;
  fTotalSize: cardinal;
published
Inheritance makes this class hierarchy not as verbose as it may have been with plain "flat" classes, but it is still much less readable than the IMyCameraCallback type definition.

In both cases, such class definitions make it difficult to guess to which message matches which service. You must be very careful and consistent about your naming conventions, and uncouple your service definitions in clear name spaces.

When implementing SOA services, DDD's Ubiquitous Language tends to be polluted by the class definition (getters and setters), and implementation details of the messages-based notification: your Domain code will be tied to the message oriented nature of the Infrastructure layer. We will see below (page 621) how interface callbacks will help implementing DDD's Event-Driven pattern, in a cleaner way.

16.7.3.3. Workflow adaptation

Sometimes, it may be necessary to react to some unexpected event. The consumer may need to change the workflow of the producer, depending on some business rules, an unexpected error, or end-user interaction.

By design, message-based implementations are asynchronous, and non-blocking: messages are sent and stored in a message broker/bus, and its internal processing loop propagates the messages to all subscribers. In such an implementation, there is no natural place for "reverse" feedback messages.

A common pattern is to have a dedicated set of "answer/feedback" messages, to notify the service providers of a state change - it comes with potential race conditions, or unexpected rebound phenomenons, for instance when you add a node to an existing event-driven system.

Another solution may be to define explicit rules for service providers, e.g. when the service is called. You may define a set of workflows, injected to the provider/bus service at runtime. It will definitively tend to break the Single Responsibility Principle (page 389), and put logic in the infrastructure layer.

On the other hand, since mORMot's callbacks are true interface methods, they may return some values (as a function result or a var/out parameter). On the server side, such callbacks will block and wait for the client end to respond.

So by writing an additional method like:

```cpp
IMyCameraCallback = interface(IInvokable)
... function ShouldRetryIfBusy(const Camera: TCameraID; const Picture: TPictureID): boolean;
...```
... you will be able to implement any needed complex workflow adaptation, in real time. The server side code will still be very readable and efficient, with no complex plumbing, wait queue or state machine to set up.

**16.7.3.4. From interfaces comes abstraction and ease**

As an additional benefit, integration with the Delphi language is clearly implementation agnostic: you are not even tied to use the framework, when working with such interface type definitions. In fact, this is a good way of implementing callbacks conforming to SOLID design principles (page 389) on the server side, and let the mORMot framework publish this mechanism in a client/server way, by using WebSockets, only if necessary.

The very same code could be used on the server side, with no transmission nor marshalling overhead (via direct interface instance calls), and over a network, with optimized use of resource and bandwidth (via "fake" interface calls, and binary/JSON marshalling over TCP/IP).

On the server side, your code - especially your Domain code - may interact directly with the lower level services, defined in the Domain as interface types, and implemented in the infrastructure layer. You may host both Domain and Infrastructure code in a single server executable, with direct assignment of local class instance as callbacks. This will minimize the program resources, in both CPU and memory terms - which is always a very valuable goal, for any business system.

You may be able to reuse your application and business logic in a stand-alone application, with similar direct calls from the UI to the application interface. On need, the interface variable may point to a remote mORMot server, without touching VCL/FMX code.

Last but not least, using an interface will help implementing the whole callback mechanism using Stubs and mocks (page 407), e.g. for easy unit testing via Calls tracing (page 415). You may also write your unit tests with real local callback class instances, which will be much easier to debug than over the whole client/server stack. Once you identified a scenario which fails the system, you could reproduce it with a dedicated test, even in an aggressive multi-threaded way, then use the debugger to trace the execution and identify the root cause of the issue.
16.8. Implementation details

16.8.1. Error handling

Usually, in Delphi applications (like in most high-level languages), errors are handled via exceptions. By default, any Exception raised on the server side, within an interface-based service method, will be intercepted, and transmitted as an HTTP error to the client side, then a safe but somewhat obfuscated EInterfaceFactoryException will be raised, containing additional information serialized as JSON.

You may wonder why exceptions are not transmitted and raised directly on the client side, as if they were executed locally.

In fact, Exceptions are not value objects, but true class instances, with some methods and potentially internal references to other objects. Most of the time, they are tied to a particular execution context, and even some low-level implementation details. A Delphi exception is even something very specific, and will not be easily converted into e.g. a JavaScript, Java or C# exception.

In practice, re-creating and raising an instance of the same Exception class which occurred on the server side will induce a strong dependency of the client code towards the server implementation details. For instance, if the server side raises a ESQLDBOracle exception, translating it on the other end will link your client side with the whole SynDBOracle.pas unit, which certainly not worth it. The ESQLDBOracle exception, by itself, contains a link to an Oracle statement instance, which will be lost when transmitted over the wire. Some client platforms (e.g. mobile or AJAX) do not even have any knowledge of what an Oracle database is...

As such, exception are not good candidate on serialization, and transmission per value, from the server side to the client side. We will NOT be in favor of propagating exceptions to the client side.

This is why exceptions should better be intercepted on the server side, with a try .. except block within the service methods, then converted into low level DTO types, specific to the service, then explicitly transmitted as error codes to the client.

The first rule is that raising exception should be exceptional - as its name states: exceptional. I mean, service code should not raise an exception in normal execution, even in case of wrong input. For instance, a wrong input parameter should lead into an application level error, transmitted as an enumeration item and/or some additional (probably text) information, but the business logic should never raise any exception. Only in case of low-level unexpected event (e.g. a SQL level failure, a GPF or Access Violation, a communication error with another trusted internal service), the server side may enter in panic mode, and raise an exception. Remember that exceptions are intercepted by SynLog.pas and can be easily logged by our below (page 631): you will be able to identify the execution context, and find a full stack trace of the issue. But most common errors should be handled at business logic level, even defined in each service layers.

In practice, you may use an enumerate, in conjunction with a variant for additional structured information (as a string or a more complex TDocVariant), to transmit an error to the client side. You may define dedicated types at every layer, e.g. with interface types for Domain services, or Application services.

See for instance how ICQRSService, and its associated TCQRSResult enumeration, are defined in mORMotDDD.pas:

type
TCQRSResult =
  (cqrsSuccess, cqrsSuccessWithMoreData,
   cqrsUnspecifiedError, cqrsBadRequest,
The first `cqrsSuccess` item of the TCQRSResult enumerate will be the default one (mapped and transmitted to a 0 JSON number), so in case of any stub or mock of the interfaces, fake methods will return as successful, as expected - see *Stubs and mocks* (page 407).

When any exception is raised in a service method, a TCQRSResult enumeration value can be returned as result, so that error will be transmitted directly:

```pascal
function TDDDMonitoredDaemon.Stop(out Information: variant): TCQRSResult;
...
begin
  CqrsBeginMethod(qaNone,result);
  try
    ....
  except
    on E: Exception do
      CqrsSetResult(E,cqrsInternalError,result);
  end;
end;
```

But such exception should be exceptional, as we already stated.

The `mORMotDDD.pas` unit defines, in the TCQRSQueryObject abstract class, some protected methods to handle errors and exceptions as expected by ICQRSService. For instance, the `TCQRSQueryObject.CqrsSetResult()` method will set `result := cqrsInternalError` and serialize the `E: Exception` within the internal variant used for additional error, ready to be retrieved using `ICQRSService.GetLastErrorInfo`.

Exceptions are very useful to interrupt a process in case of a catastrophic failure, but they are not the best method for transmitting errors over remote services. Some newer languages (e.g. Google's Go), will even not define any exception type at language or RTL level, but rely on returned values, to transmit the errors in between execution contexts - see [https://golang.org/doc/faq#exceptions](https://golang.org/doc/faq#exceptions)... in our client-server error handling design, we followed the same idea.

## 16.8.2. Security

As stated in the features grid of *Client-Server services via interfaces* (page 419), a complete security pattern is available when using client-server services. In a *Service-Oriented Architecture (SOA)* (page 90), securing messages between clients and services is essential to protecting data.

Security is implemented at several levels, following the main security patterns of `mORMot` - see below (page 544):
- *Process safety*, mainly for communication stream - e.g. when using HTTPS protocol at the *Client-Server process* (page 318), or a custom cypher within HTTP content-encoding;
- At RESTful / URI *authentication* level - see below (page 546) about `Session`, `Group` and `User` notions;
- Via *authorization* at interface or method (service/operation) level to allow or forbid a given operation.
Let us discuss the two last points now (*authentication* and *authorization*).

By default, the settings are the following for interface-based services:
- All services (i.e. all interfaces) expect one *authentication scheme* to be validated (at least TSQLRestServerAuthenticationWeak), i.e. a light session to have been initiated by the client - in short, explicit authentication is mandatory;
- All operations (i.e. all methods) are allowed to execution - in short, authorization is enabled but opened.

You can change these settings on the server side (it's an implementation detail - so it does not make any sense to tune it on the client side) via the TServiceFactoryServer instance corresponding to each interface. You can access those instances e.g. from the TSQLRestServer.Services property.

To disable the whole service / interface need of authentication, you can use the ByPassAuthentication property of the TServiceFactoryServer instance corresponding to a given interface. It may be useful e.g. for simple web services which do not expose any sensitive data (e.g. a service catalog, or a service returning public information or even HTML content).

Then, to tune the authorization process at operational (method) level, TServiceFactoryServer provides the following methods to change the security policy for each interface:
- AllowAll() and Allow() to enable methods execution globally;
- DenyAll() and Deny() to disable methods execution globally;
- AllowAllByID() and AllowByID() to enable methods execution by Group IDs;
- DenyAllByID() and DenyByID() to disable methods execution by Group IDs;
- AllowAllByName() and AllowByName() to enable methods execution by Group names;
- DenyAllByName() and DenyByName() to disable methods execution by Group names.

The first four methods will affect everybody. The next *ByID()* four methods accept a list of *authentication Group* IDs (i.e. TSQLAuthGroup.ID values), where as the *ByName()* methods will handle TSQLAuthGroup.Ident property values.

In fact, the execution can be authorized for a particular group of authenticated users. Your service can therefore provide some basic features, and then enables advanced features for administrators or supervisors only. Since the User / Group policy is fully customizable in our RESTful authentication scheme - see below (page 546), mORMot provides a versatile and inter-operable security pattern.

Here is some extract of the supplied regression tests:

```pascal
S := fClient.Server.Services['Calculator'] as TServiceFactoryServer;
Test([1,2,3,4,5],'by default, all methods are allowed');
S.AllowAll;
Test([1,2,3,4,5],'AllowAll should change nothing');
S.DenyAll;
Test([], 'DenyAll will reset all settings');
S.AllowAll;
Test([1,2,3,4,5],'back to full access for everybody');
S.DenyAllByID([GroupID]);
Test([], 'our current user shall be denied');
S.AllowAll;
Test([1,2,3,4,5],'restore allowed for everybody');
S.DenyAllByID([GroupID+1]);
Test([1,2,3,4,5],'this group ID won''t affect the current user');
S.DenyByID(['Add'], GroupID);
Test([2,3,4,5],'exclude a specific method for the current user');
S.DenyByID(['totext'], GroupID);
Test([2,3,5],'exclude another method for the current user');
(....)
```
In the above regression tests code, the Test() local procedure is used to validate the corresponding methods of ICalculator according to a set of method indexes (1=Add, 2=Multiply, 3=Subtract, 4=ToText...).

In this code, the GroupID value was retrieved as such:

```
GroupID := fClient.MainFieldID(TSQLAuthGroup,'User');
```

And the current authenticated user on the client side has been defined to be a member of the 'User' group:

```
fClient.SetUser('User','synopse'); // default user for Security tests
```

Since TSQLRestServer.ServiceRegister and TSQLRestServer.ServiceDefine methods return the first created TServiceFactoryServer instance, and since all Allow* / AllowAll* / Deny* / DenyAll* methods return also a TServiceFactoryServer instance, you can use some kind of "fluent interface" in your code to set the security policy, as such:

```
Server.ServiceDefine(TServiceCalculator,[ICalculator],sicShared).
    DenyAll.AllowAllByName(["Supervisor"]);
```

This will allow access to the ICalculator methods only for the Supervisor group of users.

### 16.8.3. Implementation class types

Most of the time, your implementation class will inherit from TInterfacedObject. As stated above, you could in fact inherit from any plain Delphi class: the only condition is that it implements the expected interface, and has a GUID.

But if you need a special process to take place during the class instance initialization, you can inherit from the TInterfacedObjectWithCustomCreate class, which provides the following virtual constructor, ready to be overridden with your customized initialization:

```
TInterfacedObjectWithCustomCreate = class(TInterfacedObject)
public
  /// this virtual constructor will be called at instance creation
  constructor Create; virtual;
end;
```

But from the SOA point of view, it could make sense to use a dedicated method with proper parameters to initialize your instance, e.g. in you are in sicClientDriven execution mode. See in Enhanced sample: remote SQL access (page 439) some sample code implementing a IRemoteSQL service, with a dedicated Connect() method to be called before all other methods to initialize a sicClientDriven instance.

### 16.8.4. Server-side execution options (threading)

When a service is registered on the server side, some options can be defined in order to specify its execution details, using the TServiceFactoryServer.SetOptions() method.

By default, service methods are called within the thread which received them. That is, when hosted by multi-threaded server instances (e.g. TSQLite3HttpServer or TSQLRestServerNamedPipeResponse), the method context can be re-entrant - unless it has been defined with sicSingle or sicPerThread instance lifetime modes. It allows better response time and CPU use, but drawback is that the method implementation shall be thread-safe. This is the technical reason why service implementation methods have to handle multi-threading safety carefully, e.g. by using Safe locks for multi-thread applications (page 123) on purpose.

The following execution options are available:
<table>
<thead>
<tr>
<th>TServiceMethodOptions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>none (default)</td>
<td>All methods are re-entrant and shall be coded to be thread-safe</td>
</tr>
<tr>
<td>optExecLockedPerInterface</td>
<td>Each interface will be protected/locked by its own mutex</td>
</tr>
<tr>
<td>optExecInMainThread</td>
<td>Methods will be executed in the process main thread</td>
</tr>
<tr>
<td>optFreeInMainThread</td>
<td>Interface will be released in the process main thread</td>
</tr>
<tr>
<td>optExecInPerInterfaceThread</td>
<td>Each interface will execute its methods in its own thread</td>
</tr>
<tr>
<td>optFreeInPerInterfaceThread</td>
<td>Each interface will be freed in its own thread</td>
</tr>
</tbody>
</table>

Of course, SetOption() accepts an optional list of method names, if you want to tune the execution at the method level.

Setting optExecLockedPerInterface option will lock the specified method(s) execution at the interface level. That is, it won't be possible to have two methods of the same interface be executed concurrently. This option uses a TRTLCriticalSection mutex, so is at the same time safe and using very little resources. But it won't guaranty that the method execution will always take place in the same thread: so if you need some per-thread initialization/finalization (e.g. for COM objects), you should better use the other options.

Setting optExecInMainThread option will force the specified method(s) to be called within a RunningThread.Synchronize() call - it can be used e.g. if your implementation rely heavily on COM objects, or if you want to ensure that your code will work correctly, without the need to worry about thread safety, which can be quite difficult to deal with. The optFreeInMainThread option will also ensure that the service class instance will be released in the main thread (i.e. its Free method called via Synchronize). Since the main thread will be used by all interfaces, it could result into an execution bottleneck.

Setting optExecInPerInterfaceThread option will force the specified method(s) to be called within a thread (to be more precise, a TSynBackgroundThreadSQLRestServerProcedure class, which will notify the TSQSLSQLRestServer for the thread context) dedicated to the interface. An associated optFreeInPerInterfaceThread option will also ensure that the service class instance will be released in the same thread: it is pretty convenient to use this threading model, for instance if you want to maintain a dedicated SynDB.pas-based database connection, or initialize some COM objects.

For instance, if you want all the methods of your TServiceCalculator class to be executed in the main thread, you can define:

```pascal
Server.ServiceDefine(TServiceCalculator,[ICalculator],sicShared).
SetOptions([],[optExecInMainThread]);
```

Or if only the TServiceCalculator.Add method has to be protected, you can write:

```pascal
Server.ServiceDefine(TServiceCalculator,[ICalculator],sicShared).
SetOptions(['Add'],[optExecInMainThread]);
```

In fact, the SetOptions() method follows a call signature similar to the one used for defining the service security.

For best performance, you may define your service methods be called without any locking, but rely on some convenient classes defined in SynCommons.pas - as the TAutoLocker class or the TLockedDocVariant kind of storage, for efficient multi-thread process.

A similar thread safety concern also applies to MVVM methods - see below (page 529).
16.8.5. Audit Trail for Services

We have seen previously how the ORM part of the framework is able to provide an Audit Trail for change tracking (page 178). It is a very convenient way of storing the change of state of the data. On the other side, in any modern SOA solution, data is not at the center any more, but services. Sometimes, the data is not stored within your server, but in a third-party Service-Oriented Architecture (SOA) (page 90). Being able to monitor the service execution of the whole system becomes sooner or later mandatory. Our framework allows to create an Audit Trail of any incoming or outgoing service operation, in a secure, efficient and automated way.

16.8.5.1. When logging is not enough

By default, any interface-based service process will be logged by the framework - see below (page 641) - in dedicated sllServiceCall and sllServiceReturn log levels. You may see output similar to the following:

```
18:03:18 Enter   mORMot.TSQLRestServerFullMemory(024500A0).URI(POST root/DomUserQuery.SelectByLogonName/1 inlen=7)
18:03:18 Service call mORMot.TSQLRestServerFullMemory(024500A0)
   DomUserQuery.SelectByLogonName["979"]
18:03:18 Server mORMot.TSQLRestServerFullMemory(024500A0) POST root/DomUserQuery.SelectByLogonName SOA-Interface -> 200 with outlen=21 in 16 us
18:03:18 Service return mORMot.TSQLRestServerFullMemory(024500A0) {"result":[0],"id":1}
18:03:18 Leave   00.000.017
```

The above lines match the execution of the following method, as defined in dddDomUserCQRS.pas:

```pascal
IDomUserQuery = interface (ICQRSService)
  ['{198C01D6-5189-4B74-AAF4-C322237D7D53}']
  // will select a single TUser from its logon name
  // - then use Get() method to retrieve its content
  function SelectByLogonName(const aLogonName: RawUTF8): TCQRSResult;
...
```

The actual execution was:

```
IDomUserQuery.SelectByLogonName('979') -> cqrsSuccess
```

Here cqrsSuccess is the first item of the enumeration result, returned as an integer JSON value "result":0 by the method:

```pascal
TCQRSResult = (cqrsSuccess, cqrsSuccessWithMoreData,
        cqrsUnspecifiedError, cqrsBadRequest, cqrsNotFound,
...
```

This detailed log (including micro-second timing on the "Leave" rows) is very helpful for support, especially to investigate any error occurring on a production server. But it will not be enough (or on the contrary provide "too much information" which "kills the information") to monitor the higher level of the process, especially on a server with a lot of concurrent activity.

16.8.5.2. Tracing Service Methods

The framework allows to optionally store each SOA method execution in a database, with the input and output parameters, and accurate timing.

You could enable this automated process:
- Either at service level, using TServiceFactoryServer.SetServiceLog();
- Or for all services of a TSQLRestServer.ServiceContainer instance, via TServiceContainerServer.SetServiceLog().

For instance, you may enable it for a whole REST server:
This single command will create an Audit Trail with all service calls made on aRestSOAServer to the TSQLRecordServiceLog ORM class of aRestLogServer. Keeping a dedicated REST server for the log entries will reduce the overhead on the main server, and ease its maintenance.

Actual storage takes place within a class inheriting from TSQLRecordServiceLog:

```pascal
(aRestSOAServer.ServiceContainer as TServiceContainerServer).SetServiceLog(aRestLogServer,TSQLRecordServiceLog);
```

The ORM will therefore store the following table on its database:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>TID</td>
</tr>
<tr>
<td>Input</td>
<td>variant</td>
</tr>
<tr>
<td>Method</td>
<td>RawUTF8</td>
</tr>
<tr>
<td>MicroSec</td>
<td>integer</td>
</tr>
<tr>
<td>Output</td>
<td>variant</td>
</tr>
<tr>
<td>Session</td>
<td>integer</td>
</tr>
<tr>
<td>Time</td>
<td>TModTime</td>
</tr>
<tr>
<td>User</td>
<td>integer</td>
</tr>
</tbody>
</table>

As you can see, all input and output parameters are part of the record, as two TDocVariant instances. Since they are stored as JSON/TEXT, you could perform some requests directly on their content, especially if actual storage take place in a MongoDB database: you may even use dedicated indexes on the parameter values, and/or run advanced map/reduce queries. You can use optNoLogInput or optNoLogOutput settings with TInterfaceFactory.SetOptions() to hide all input or output parameters values, or define some value types as containing Sensitive Personal Information (SPI), using TInterfaceFactory.RegisterUnsafeSPIType.

Since very accurate timing, with a micro-second resolution, is part of the information, you will be able to make filtering or advanced statistics using simple SQL clauses. It has never been easier to monitor your SOA system, and identify potential issues. You may easily extract this information from your
database, and feed a real-time visual monitoring chart system, for instance. Or identify and spy unusual execution patterns (e.g. unexpected timing or redounding error codes), which will match some SQL requests: those SQL statements may be run automatically on a regular basis, to prevent any problem before it actually happen.

16.8.5.3. Tracing Asynchronous External Calls

Sometimes, your server may be the client of another process. In an SOA environment, you may interface with a third-party REST service for an external process, e.g. sending a real-time notification.

On the REST client instance, you can execute the TServiceFactoryClient.SendNotifications() method for a given service:

```
//NotificationClientService.SendNotifications(aServicesLogRest,
TSQLRecordServiceNotifications, fSettings.NotificationsRetrySeconds);
```

This single command will create an Audit Trail with all notification calls sent to aNotificationClientService, in the TSQLRecordServiceNotifications ORM class of aServicesLogRest.

You may use the following TSQLRecordServiceNotifications class:

```
TSQLRecordServiceNotifications = class(TSQLRecordServiceLog)
...
published
  /// when this notification has been sent
  /// - equals 0 until it was actually notified
  property Sent: TTimeLog read fSent write fSent;
end;
```

Which will be stored in the following table:

|----------|-----------------|----------------|------------------|-------------------|-----------------|-----------------|----------------|--------------|

**ServiceNotifications Record Layout**

The additional Sent property will contain the TTimeLog time-stamp on which the notification will have taken place.

In fact, all methods executed via this notification service will now be first stored in this table, then the remote HTTP notifications will take place asynchronously in the background. Transmission will be in order (first-in-first-out), and in case of any connection problem (e.g. the remote server not returning a 200 HTTP SUCCESS status code), it won't move to the next entry, and will retry after the NotificationsRetrySeconds period, as supplied to the SendNotifications() method.

Of course, you may define your own sub-class, to customize the destination Audit Trail table:

```
type
  TSQLMyNotifications = class(TSQLRecordServiceNotifications);
```

Thanks to those TSQLRecordServiceLog classes, high-level support and analysis has never become easier. The actual implementation of those features has been tuned to minimize the impact on main
performance, by using e.g. delayed write operations via *BATCH sequences for adding/updating/deleting records* (page 350), or a dedicated background thread for the asynchronous notification process.

### 16.8.6. Transmission content

All data is transmitted as JSON arrays or objects, according to the requested URI.

We'll discuss how data is expected to be transmitted, at the application level.

#### 16.8.6.1. Request format

As stated above, there are several available modes of routing, defined by a given class, inheriting from `TSQLRestServerURIContext`:

![Routing via TSQLRestServerURIContext classes hierarchy](image)

The corresponding description may be:

<table>
<thead>
<tr>
<th></th>
<th>TSQLRestRoutingREST</th>
<th>TSQLRestRoutingJSON_RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mode</strong></td>
<td>RESTful</td>
<td>JSON-RPC</td>
</tr>
<tr>
<td><strong>Default</strong></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>URI scheme</strong></td>
<td><code>/Model/Interface.Method[/ClientDrivenID]</code></td>
<td><code>/Model/Interface</code></td>
</tr>
<tr>
<td></td>
<td>or <code>/Model/Interface/Method[/ClientDrivenID]</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ optional URI-encoded params</td>
<td></td>
</tr>
<tr>
<td><strong>Body content</strong></td>
<td>JSON array of parameters or void if parameters</td>
<td>`{&quot;method&quot;:&quot;MethodName&quot;, &quot;params&quot;:[]}</td>
</tr>
<tr>
<td></td>
<td>were encoded at URI</td>
<td><code>[,&quot;id&quot;:ClientDrivenID]</code></td>
</tr>
<tr>
<td><strong>Body content</strong></td>
<td>JSON object of parameters or void if parameters</td>
<td>`{&quot;method&quot;:&quot;MethodName&quot;, &quot;params&quot;:[]}</td>
</tr>
<tr>
<td>(alternative)</td>
<td>were encoded at URI</td>
<td><code>[,&quot;id&quot;:ClientDrivenID]</code></td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td>RESTful authentication for each method</td>
<td>RESTful authentication for the</td>
</tr>
<tr>
<td></td>
<td>or for the whole service (interface)</td>
<td>whole service (interface)</td>
</tr>
<tr>
<td><strong>Speed</strong></td>
<td>10% faster</td>
<td>10% slower</td>
</tr>
</tbody>
</table>

Most of the time, the input parameters will be transmitted as a JSON array of values, following the exact order of `const / var` method parameters. As an alternative, a JSON object storing the input parameters by name will be accepted. This will be slightly slower than a JSON array of parameters, but could be handy, depending on the client side. Last but not least, TSQLRestRoutingREST is able to decode parameters encoded at URI level, as most regular historic HTTP requests.
The routing to be used is defined globally in the TSQLRest.ServiceRouting property, and should match on both client and server side, of course. By design, you should never assign the abstract TSQLRestServerURIContext to this property.

The TSQLRestServerURIContext abstract class defines the following methods, which will be overridden by inherited implementations to reflect the expected behavior on all aspects of the RESTful routing and transmission:

```cpp
TSQLRestServerURIContext = class
protected
  ... // retrieve RESTful URI routing
  function URIDecodeREST: boolean; virtual;
  // retrieve method-based SOA URI routing with optional RESTful mode
  procedure URIDecodeSOAByMethod; virtual;
  // retrieve interface-based SOA
  procedure URIDecodeSOAByInterface; virtual; abstract;
  // process authentication
  function Authenticate: boolean; virtual;
  // direct launch of a method-based service
  procedure ExecuteSOAByMethod; virtual;
  // direct launch of an interface-based service
  procedure ExecuteSOAByInterface; virtual; abstract;
  // handle GET/LOCK/UNLOCK/STATE verbs for ORM/CRUD process
  procedure ExecuteORMGet; virtual;
  // handle POST/PUT/DELETE/BEGIN/END/ABORT verbs for ORM/CRUD process
  procedure ExecuteORMWrite; virtual;
  ...
```

Most of the time, the supplied TSQLRestRoutingREST and TSQLRestRoutingJSON_RPC classes will meet your requirements.

16.8.6.1.1. REST mode

16.8.6.1.1. Parameters transmitted as JSON array

In the default TSQLRestRoutingREST mode, both service and operation (i.e. interface and method) are identified within the URI. And the message body is a standard JSON array of the supplied parameters (i.e. all const and var parameters).

Here is typical request for ICalculator.Add:

```plaintext
POST /root/Calculator.Add
    (...) [1,2]
```

Here we use a POST verb, but the framework will also allows other methods like GET, if needed (e.g. from a regular browser). The pure Delphi client implementation will use only POST.

For a sicClientDriven mode service, the needed instance ID is appended to the URI:

```plaintext
POST /root/ComplexNumber.Add/1234
    (...) [20,30]
```

Here, 1234 is the identifier of the server-side instance ID, which is used to track the instance life-time, in sicClientDriven mode. One benefit of transmitting the Client Session ID within the URI is that it will be more secure in our RESTful authentication scheme - see below (page 546): each method (and even any client driven session ID) will be signed properly.

16.8.6.1.1.2. Parameters transmitted as JSON object
The mORMot server will also accept the incoming parameters to be encoded as a JSON object of named values, instead of a JSON array:

```
POST /root/Calculator.Add
(...)  
{"n1":1,"n2":2}
```

Of course, order of the values is not mandatory in a JSON object, since parameters will be lookup by name. As a result, the following request will be the same as the previous one:

```
POST /root/Calculator.Add
(...)  
{"n2":2,"n1":1}
```

For a sicClientDriven mode service, the needed instance ID is appended to the URI:

```
POST /root/ComplexNumber.Add/1234
(...)  
{"aReal":20,"aImaginary":30}
```

In some cases, naming the parameters could be useful, on the client side. But this should not be the default, since it will be slightly slower (for parsing and checking the names), and use more bandwidth at transmission.

Any missing parameter in the incoming JSON object will be replaced by its default value. For instance, the following will run IComplexNumber.Add(0,2):

```
POST /root/Calculator.Add
(...)  
{"n2":2}
```

Any unknown parameter in the incoming JSON object will just be ignored. It could be handy, if you want to transmit some generic execution context (e.g. a global “data scope” in a MVC model), and let the service use only the values it needs.

```
POST /root/ComplexNumber.Add/1234
(...)  
{"Session":"1234","aImaginary":30,"aReal":20,"UserLogged":"Nikita"}
```

Of course, the extra values will consume some bandwidth for nothing, but the process cost on the server side will be negligible, since our implementation will just ignore those unexpected properties, without allocating any memory for them.

### 16.8.6.1.1.3. Parameters encoded at URI level

In this TSQLRestRoutingREST mode, the server is also able to retrieve the parameters from the URI, if the message body is left void. This is not used from a Delphi client (since it will be more complex and therefore slower), but it can be used for a client, if needed:

```
POST root/Calculator.Add?+%5B+1%2C2+%5D
GET root/Calculator.Add?+%5B+1%2C2+%5D
```

In the above line, +%5B+1%2C2+%5D will be decoded as [1,2] on the server side. In conjunction with the use of a GET verb, it may be more suitable for a remote AJAX connection.

As an alternative, you can encode and name the parameters at URI level, in a regular HTML fashion:

```
GET root/Calculator.Add?n1=1&n2=2
```

Since parameters are named, they can be in any order. And if any parameter is missing, it will be replaced by its default value (e.g. 0 for a number or '' for a string).

This may be pretty convenient for simple services, consumed from any kind of client.
Note that there is a known size limitation when passing some data with the URI over HTTP. Official RFC 2616 standard advices to limit the URI size to 255 characters, whereas in practice, it sounds safe to transmit up to 2048 characters within the URI. If you want to get rid of this limitation, just use the default transmission of a JSON array as request body.

As an alternative, the URI can be written as `/RootName/InterfaceName/MethodName`. It may be more RESTful-compliant, depending on your client policies. The following URIs will therefore be equivalent to the previous requests:

```plaintext
POST /root/Calculator/Add
POST /root/ComplexNumber/Add/1234
POST root/Calculator/Add?%5B%5B1%2C2%5D
GET root/Calculator/Add?%5B%5B1%2C2%5D
GET root/Calculator/Add?n1=1&n2=2
```

From a Delphi client, the `/RootName/InterfaceName.MethodName` scheme will always be used.

**16.8.6.1.4. Sending a JSON object**

By default, the mORMot client will send all values, transmitted as a JSON array without any parameter name, as we have seen:

```plaintext
POST /root/Calculator.Add
(...)
[1,2]
```

But if `TServiceFactoryClient.ParamsAsJSONObject` is set to true, the transmitted values from the client side will be encoded as a JSON object:

```plaintext
POST /root/Calculator.Add
(...)
{"n1":1,"n2":2}
```

This may help transmitting some values to a non-mORMot server, in another format, for a given service.

**16.8.6.1.5. Sending raw binary**

If your purpose is to upload some binary data, `RawByteString` and `TSQLRawBlob` input parameters will by default be transmitted as Base64 encoded JSON text.

You may define *Client-Server services via methods* (page 373) to transmit raw binary, without the Base64 encoding overhead. It would allow low-access to the input content type and encoding, even with multi-part file upload from HTTP.

As an alternative, if you use default `TSQLRestRoutingREST` routing, and defined a single `RawByteString` or `TSQLRawBlob` input parameter, it will be processed as a raw POST with binary body defined with mime-type `application/octet-stream`. This may be more optimized for remote access over the Internet.

**16.8.6.1.2. JSON-RPC**

**16.8.6.1.2.1. Parameters transmitted as JSON array**

If `TSQLRestRoutingJSON_RPC` mode is used, the URI will define the interface, and then the method name will be inlined with parameters, e.g.

```plaintext
POST /root/Calculator
(...)
{"method":"Add","params": [1,2],"id":0}
```
Here, the "id" field can be not set (and even not existing), since it has no purpose in sicShared mode.

For a sicClientDriven mode service:

```json
POST /root/ComplexNumber
    {"method":"Add","params":[20,30],"id":1234}
```

### 16.8.6.1.2.2. Parameters transmitted as JSON object

As an alternative, you may let the values be transmitted as a JSON object containing the named parameters values, instead of a JSON array:

```json
POST /root/Calculator
    {"method":"Add","params":{"n1":1,"n2":2},"id":0}
```

Here, the same rules applies than in TSQLRestRoutingREST mode:
- Any missing parameter will be replaced by its default value;
- Properties order is not sensitive anymore;
- Unexpected parameters will just be ignored.

Note that by definition, TSQLRestRoutingJSON_RPC mode is not able to handle URI-encoded parameters. In fact, the JSON-RPC mode expects the URI to be used only for identifying the service, and have the whole execution context transmitted as body.

### 16.8.6.1.3. REST mode or JSON-RPC mode?

For a standard mORMot Delphi client, or any supported Cross-Platform client - see below (page 481) - TSQLRestRoutingREST is preferred. The supplied libraries, even for Smart Mobile Studio, fully implement this routing scheme. It is the faster, safer and most modular mode available.

In practice, TSQLRestRoutingJSON_RPC mode has been found to be a little bit slower. Since the method name will be part of the URI, the signature will have a bigger extent than in JSON-RPC mode, so it will be more secure. Its ability to retrieve URI-encoded parameters could be also useful, e.g. to server some dynamic HTML pages in addition to the SOA endpoints, with proper HTTP caching abilities.

Of course, TSQLRestRoutingJSON_RPC mode may be used as an alternative, depending on the client expectations, and technology limitations, e.g. if your client expect a JSON-RPC compatible communication.

It's up to you to select the right routing scheme to be used, depending on your needs.

### 16.8.6.2. Response format

#### 16.8.6.2.1. Standard answer as JSON object

#### 16.8.6.2.1.1. JSON answers

The framework will always return the data in the same format, whatever the routing mode used.

Basically, this is a JSON object, with one nested "result": property, and the client driven "id": value (e.g. always 0 in sicShared mode):
POST /root/Calculator.Add
(...) [1,2]

will be answered as such:

```
{"result": [3]}
```

For a sicClientDriven, sicPerSession, sicPerUser, sicPerGroup or sicPerThread mode service, the answer will contain an additional "id":... member, which will identify the corresponding session:

```
{"result": [3], "id": 1234}
```

In sicSingle and sicShared modes, the "id":0 member is just not emitted.

The result JSON array contains all var and out parameters values (in their declaration order), and then the method main result.

For instance, here is a transmission stream for a ICalculator.ComplexCall request in TSQLRestRoutingREST mode:

```
POST /root/Calculator.ComplexCall
([^..]) [[128722814,1231886296], ["one","two","three"], ["ABC","DEF","GHIJK"], "i[8]bAAAAnAA[AA][AAA][AA]CNE01x2ZXcbGl1xFNRTGl02Tnc3Xk1XFR1c3RTUWwzLmW4ZQ==", "i[8]xow1EdX3bUKyWJoJ"]
```

will be answered as such:

```
```

It matches the var / const / out parameters declaration of the method:

```
function ComplexCall(const Ints: TIntegerDynArray; Strs1: TRawUTF8DynArray; const Rec1: TVirtualTableModuleProperties; var Str2: TWideStringDynArray; const Rec2: TSQLRestCacheEntryValue): TSQLRestCacheEntryValue;
```

And its implementation:

```
function TServiceCalculator.ComplexCall(const Ints: TIntegerDynArray; Strs1: TRawUTF8DynArray; var Str2: TWideStringDynArray; const Rec1: TVirtualTableModuleProperties; var Rec2: TSQLRestCacheEntryValue): TSQLRestCacheEntryValue;
var i: integer;
begin
  result := Rec2;
  result.JSON := StringToUTF8(Rec1.FileExtension);
  i := length(Str2);
  SetLength(Str2, i + 1);
  Str2[i] := UTF8ToWideString(RawUTF8ArrayToCSV(Strs1));
  inc(Rec2.ID);
  dec(Rec2.Timestamp);
  Rec2.JSON := IntegerDynArrayToCSV(Ints, length(Ints));
end;
```

Note that TIntegerDynArray, TRawUTF8DynArray and TWideStringDynArray values were marshaled as JSON arrays, whereas complex records (like TSQLRestCacheEntryValue) have been Base64 encoded.

If you want to transmit some binary blob content, consider using a RawByteString kind of parameter, which will transmit a Base64-encoded JSON text on the wire.

The framework is able to handle class instances as parameters, for instance with the following interface, using a TPersistent child class with published properties (it will be the same for TSQLRecord ORM instances):
type TComplexNumber = class(TPersistent)
  private
    fReal: Double;
    fImaginary: Double;
  public
    constructor Create(aReal, aImaginary: double); reintroduce;
  published
    property Real: Double read fReal write fReal;
    property Imaginary: Double read fImaginary write fImaginary;
end;

IComplexCalculator = interface(ICalculator)
  ['{8D0F3B39-056B-4488-A616-9B6CF8D4DEB7}']
  /// purpose of this unique method is to subtract two complex numbers
  /// - using class instances as parameters
  procedure Subtract(n1,n2: TComplexNumber; out Result: TComplexNumber);
end;

As stated above, it is not possible to return a class as a result of a function (who will be responsible of handling its life-time?). So in this method declaration, the result is declared as out parameter.

During the transmission, published properties of TComplexNumber parameters will be serialized as standard JSON objects within the "result":[... ] JSON array:

```
POST root/ComplexCalculator.Subtract
(...) [["Real":2,"Imaginary":3],{"Real":20,"Imaginary":30}]
```

will be answered as such:

```
{"result":{"Real":-18,"Imaginary":-27}}
```

### 16.8.6.2.1.1.2. Returning a JSON object

Note that if TServiceFactoryServer.ResultAsJSONObject is set to true, the outgoing values won't be emitted within a "result":[... ] JSON array, but via a "result":{... } JSON object, with the var/out parameter names as object fields, and "Result": for a function result:

```
{"result":{"Result":{"Real":-18,"Imaginary":-27}}}
```

The TServiceFactoryServer.ResultAsJSONObjectWithoutResult property may be used to avoid the main "Result": object.

Instead of this JSON array content, returned by default:

```
GET root/Calculator/Add?n1=1&n2=2
{"result":[3]}
```

The following JSON will be returned if TServiceFactoryServer.ResultAsJSONObject is true:

```
GET root/Calculator/Add?n1=1&n2=2
{"result":{"Result":3}}
```

Or, if TServiceFactoryServer.ResultAsJSONObjectWithoutResult is true:

```
GET root/Calculator/Add?n1=1&n2=2
...
{"Result":3}
```

All those JSON array or object contents fulfill perfectly standard JSON declarations, so can be generated and consumed directly by any AJAX client. The TServiceFactoryServer. ResultAsJSONObject option make it even easier to consume mORMot services, since all outgoing values will be named in the "result": JSON object.
16.8.6.2.1.2. Returning raw JSON content

By default, if you want to transmit a JSON content with interface-based services, using a RawUTF8 will convert it to a JSON string. Therefore, any JSON special characters (like " or \ or []) will be escaped. This will slow down the process on both server and client side, and increase transmission bandwidth.

For instance, if you define such a method:

```pascal
function TServiceRemoteSQL.Execute(const aSQL: RawUTF8; aExpectResults, aExpanded: Boolean): RawUTF8;
var res: ISQLDBRows;
begin
  if fProps=nil then
    raise Exception.Create('Connect call required before Execute');
  res := fProps.ExecuteInlined(aSQL,aExpectResults);
  if res=nil then
    result := ''
  else
    result := res.FetchAllAsJSON(aExpanded);
end;
```

The FetchAllAsJSON() method will return a JSON array content, but will be escaped as a JSON string when transmitted via a RawUTF8 variable.

A dedicated RawJSON type has been defined, and will specify to the mORMot core that the UTF-8 text is a valid JSON content, and should not be escaped.

That is, defining the method as followed will increase process speed and reduce used bandwidth:

```pascal
function TServiceRemoteSQL.Execute(const aSQL: RawUTF8; aExpectResults, aExpanded: Boolean): RawJSON;
```

See sample "16 - Execute SQL via services" for some working code using this feature.

As a consequence, using RawJSON will also make the transmitted content much more AJAX friendly, since the returned value will be a valid JSON array or object, and not a JSON string which will need JavaScript "unstringification".

16.8.6.2.1.3. Returning errors

In case of an error, the standard message object will be returned:

```
{
  "ErrorText": "Error description"
}
```

The following error descriptions may be returned by the service implementation from the server side:

<table>
<thead>
<tr>
<th>ErrorText</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method name required</td>
<td>TSQLRestRoutingJSON_RPC call without &quot;method&quot;: field</td>
</tr>
<tr>
<td>Unknown method</td>
<td>TSQLRestRoutingJSON_RPC call with invalid method name (in this mode, there is no specific message, since a JSON answer may be a valid request)</td>
</tr>
<tr>
<td>Parameters required</td>
<td>The server expect at least a void JSON array (aka []) as parameters</td>
</tr>
<tr>
<td>Unauthorized method</td>
<td>This method is not allowed with the current authenticated user group - see Security above</td>
</tr>
</tbody>
</table>
Not allowed to publish signature

The client requested the interface signature, but this has not been allowed on the server policy (see TServiceContainerServer. PublishSignature)

... instance id:? not found or deprecated

The supplied "id": parameter points to a wrong instance (in sicPerSession / sicPerUser / sicPerGroup mode)

ExceptionClass: Exception
Message (with 500 Internal Server Error)

An exception was raised during method execution

On the client side, you may encounter the following EInterfaceFactoryException messages, starting with the generic 'Invalid fake IInterfaceName.MethodName interface call' text:

<table>
<thead>
<tr>
<th>ErrorText</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unexpected self</td>
<td>self does exist as low-level implementation detail, but is not intended to be transmitted</td>
</tr>
<tr>
<td>JSON array/object result</td>
<td>content returned from the Server was neither a JSON array nor a JSON object</td>
</tr>
<tr>
<td>expected</td>
<td></td>
</tr>
<tr>
<td>unexpected parameter &quot;....&quot;</td>
<td>the Server returned a JSON object with an unknown or invalid member name</td>
</tr>
<tr>
<td>returned object record variant</td>
<td>a returned class, record, variant, dynamic array of RawJSON value was not properly serialized</td>
</tr>
<tr>
<td>array RawJSON</td>
<td></td>
</tr>
<tr>
<td>missing or invalid value</td>
<td>a returned string or numerical value is not valid JSON content</td>
</tr>
</tbody>
</table>

16.8.6.2.2. Returning content as XML

By default, interface-based services of a mORMot server will always return a JSON array. But you may (or a JSON object, if TServiceFactoryServer.ResultAsJSONObject or ResultAsJSONObjectWithoutResult is true).

With some kind of clients (e.g. if they are made by a third party), it could be useful to return XML content instead.

Your mORMot server is able to let its interface-based services return XML context instead, or in addition to the default JSON format.

16.8.6.2.2.1. Always return XML content

If you want all methods of a given interface to return XML content instead of JSON, you can set TServiceFactoryServer.ResultAsXmlObject to true.

Instead of this JSON array content, returned by default:

```
GET root/Calculator/Add?n1=1&n2=2
...{"result":[3]}
```

The following XML will be returned if TServiceFactoryServer.ResultAsXmlObject is true:

```
GET root/Calculator/Add?n1=1&n2=2
...<result>3</result>
```
Synopse mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

<?xml version="1.0" encoding="UTF-8"?>
<result><Result>3</Result></result>

Conversion is processed from the JSON content generated by the mORMot kernel, via a call to
JSONBufferToXML() function, which performs the XML generation without almost no memory
allocation. So only a slightly performance penalty may be noticed (much faster in practice than most
node-based XML producers available).
One drawback of using this TServiceFactoryServer.ResultAsXMLObject property is that your
regular Delphi or AJAX clients won't be able to consume the service any more, since they expect JSON
content.
If you want your service to be consumed either by XML and JSON, you will need two services. You can
therefore define a dedicated interface to return XML, and then register this interface to return only
XML:
type
ICalculator = interface(IInvokable)
['{9A60C8ED-CEB2-4E09-87D4-4A16F496E5FE}']
/// add two signed 32-bit integers
function Add(n1,n2: integer): integer;
end;
ICalculatorXML = interface(ICalculator)
['{0D682D65-CE0F-441B-B4EC-2AC75E357EFE}']
end; // no additional method, just new name and GUID
TServiceCalculator = class(TInterfacedObject, ICalculator,ICalculatorXML)
public // implementation class should implement both interfaces
function Add(n1,n2: integer): integer;
end;
...
aServer.ServiceRegister(TServiceCalculator,[TypeInfo(ICalculator)],sicShared);
aServer.ServiceRegister(TServiceCalculator,[TypeInfo(ICalculatorXML)],sicShared).
ResultAsXMLObject := True;
...

There will therefore be two running service instances (e.g. here two instances of
TServiceCalculator, one for ICalculator and one for ICalculatorXML). It could be an issue, in
some cases.
And such a dedicated interface may need more testing and code on the server side, since they will be
accessible from two URIs:
GET root/Calculator/Add?n1=1&n2=2
...
{"result":{"Result":3}}

and for ICalculatorXML interface:
GET root/CalculatorXML/Add?n1=1&n2=2
...
<?xml version="1.0" encoding="UTF-8"?>
<result><Result>3</Result></result>

16.8.6.2.2.2. Return XML content on demand
As an alternative, you can let the mORMot server inspect the incoming HTTP headers, and return the
content as XML if the "Accept: " header is exactly "application/xml" or "text/xml".
You can set the TServiceFactoryServer.ResultAsXMLObjectIfAcceptOnlyXML property to enable
this HTTP header detection:
aServer.ServiceRegister(TServiceCalculator,[TypeInfo(ICalculator)],sicShared).

SAD - mORMot Framework - Rev. 1.18

Page 476 of 2552


For standard requests, the incoming HTTP header will be either void, either "Accept: */*", so will return JSON content. But if the client set either "Accept: application/xml" or "Accept: text/xml" in its header, then it will return an XML document.

Instead of this JSON content:

```plaintext
GET root/Calculator/Add?n1=1&n2=2
Accept: */*
...
"result":{"Result":3}
```

The following XML will be returned:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<result><Result>3</Result></result>
```
as it will with "text/xml":

```xml
<?xml version="1.0" encoding="UTF-8"?>
<result><Result>3</Result></result>
```

Note that the header is expected to be "Accept: application/xml" or "Accept: text/xml" as exact value.

For instance "Accept: text/html,application/xml,/*/*" won’t be detected by the server, and will return regular JSON:

```plaintext
GET root/Calculator/Add?n1=1&n2=2
Accept: text/html,application/xml,/*/*
...
"result":{"Result":3}
```

Your XML client should therefore be able to force the exact content of the HTTP "Accept:" header.

Together with parameter values optionally encoded at URI level - available with TSQLRestRoutingREST default routing scheme (see ?n1=1&n2=2 above) - it could be an useful alternative to consume mORMot services from any XML-based client.

### 16.8.6.2.3. Custom returned content

Note that even if the response format is a JSON object by default, and expected as such by our TServiceContainerClient implementation, there is a way of returning any content from a remote request.

It may be used by AJAX or HTML applications to return any kind of data, i.e. not only JSON results, but pure text, HTML or even binary content. Our TServiceFactoryClient instance is also able to handle such requests, and will save client-server bandwidth when transmitting some BLOB data (since it won’t serialized the content with Base64 encoding).

In order to specify a custom format, you can use the following TServiceCustomAnswer record type as the result of an interface function:

```delphi
TServiceCustomAnswer = record
  Header: RawUTF8;
  Content: RawByteString;
end;
```
Status: cardinal;

The Header field shall be not null (i.e. not equal to ")", and contains the expected content type header (e.g. TEXT_CONTENT_TYPE_HEADER or HTML_CONTENT_TYPE_HEADER). Then the Content value will be transmitted back directly to the client, with no JSON serialization. Of course, no var nor out parameter will be transmitted (since there is no JSON result array any more). Finally, the Status field could be overridden with a property HTML code, if the default HTTP_SUCCESS is not enough for your purpose. Note that when consumed from Delphi clients, HTTP_SUCCESS is expected to be returned by the server: you should customize Status field only for plain AJAX / web clients.

In order to implement such method, you may define such an interface:

```delphi
IComplexCalculator = interface(ICalculator)
  ['{8D0F3839-056B-4488-A616-986CF8D4DEB7}']
  function TestBlob(n: TComplexNumber): TServiceCustomAnswer;
end;
```

This may be implemented for instance as such:

```delphi
function TServiceComplexCalculator.TestBlob(n: TComplexNumber): TServiceCustomAnswer;
begin
  Result.Header := TEXT_CONTENT_TYPE_HEADER;
  Result.Content := FormatUTF8('%%', [n.Real, n.Imaginary]);
  // leave Result.Header to its default value
end;
```

This will return not a JSON object, but a plain TEXT content.

Regression tests will make the following process:

```delphi
with CC.TestBlob(C3) do begin
  Check(Header=TEXT_CONTENT_TYPE_HEADER);
  Check(Content=FormatUTF8('%%', [C3.Real, C3.Imaginary]));
end;
```

Note that since there is only one BLOB content returned, no var nor out parameters are allowed to be defined for this method. If this is the case, an exception will be raised during the interface registration step. But you can define any const parameter needed, to specify your request.

You may also be able to use this feature to implement custom UTF-8 HTML creation, setting the Header value to HTML_CONTENT_TYPE_HEADER constant, and using our fast below (page 505) for the rendering.

Remember that in TSQLRestRoutingJSON mode, you can encode any simple parameter value at URI level, to transmit your browsing context.
16.9. Comparison with WCF

Microsoft Windows Communication Foundation is the unified programming model provided by Microsoft for building service-oriented applications. See http://msdn.microsoft.com/en-us/library/dd456779..

Here is a short reference table of WCF / mORMot SOA features and implementation of the RESTful pattern.

<table>
<thead>
<tr>
<th>Feature</th>
<th>WCF</th>
<th>mORMot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal design</td>
<td>SOAP with REST integration</td>
<td>RESTful</td>
</tr>
<tr>
<td>Hosting</td>
<td>exe/service/ISS/WAS</td>
<td>in-process/exe/service</td>
</tr>
<tr>
<td>Scalability/balancing</td>
<td>up to WAS</td>
<td>by dedicated hosting</td>
</tr>
<tr>
<td>MetaData</td>
<td>WSDL+XML</td>
<td>JSON contract</td>
</tr>
<tr>
<td>Service contract</td>
<td>interface</td>
<td>interface</td>
</tr>
<tr>
<td>Data contract</td>
<td>class</td>
<td>class/record</td>
</tr>
<tr>
<td>ORM integration</td>
<td>separated</td>
<td>integrated in the model</td>
</tr>
<tr>
<td>URI definition</td>
<td>attribute-driven</td>
<td>REST/JSON-RPC convention-driven, or class-driven</td>
</tr>
<tr>
<td>Service contract</td>
<td>interface + attributes</td>
<td>interface + shared Model</td>
</tr>
<tr>
<td>Versioning</td>
<td>XML name-space</td>
<td>interface signature</td>
</tr>
<tr>
<td>Message protocol</td>
<td>SOAP/custom</td>
<td>RESTful</td>
</tr>
<tr>
<td>Messaging</td>
<td>single/duplex</td>
<td>stateless (like HTTP)</td>
</tr>
<tr>
<td>Sequence</td>
<td>attributes on methods</td>
<td>interface life time</td>
</tr>
<tr>
<td>Transactional</td>
<td>fully transactional</td>
<td>on implementation side</td>
</tr>
<tr>
<td>Instance life time</td>
<td>per call, per session, single</td>
<td>per call, per session, per user, per group, per thread, single, client-driven</td>
</tr>
<tr>
<td>Configuration</td>
<td>.config file or code</td>
<td>convention over configuration optionally tuned by code</td>
</tr>
<tr>
<td>Client access</td>
<td>Layer source should be generated</td>
<td>No layer, but direct registration</td>
</tr>
<tr>
<td>End points</td>
<td>One end-point per contract</td>
<td>Unique or shared end-point</td>
</tr>
<tr>
<td>Operation</td>
<td>synchronous/asynchronous</td>
<td>synchronous (REST)</td>
</tr>
<tr>
<td>Session</td>
<td>available (optional)</td>
<td>available (optional)</td>
</tr>
<tr>
<td>Feature</td>
<td>At Service level</td>
<td>At Communication level</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Encryption</td>
<td>at Service level</td>
<td>at communication level</td>
</tr>
<tr>
<td>Compression</td>
<td>at Service level</td>
<td>at communication level</td>
</tr>
<tr>
<td>Serialization</td>
<td>XML/binary/JSON</td>
<td>JSON/XML/custom</td>
</tr>
<tr>
<td>Communication Protocol</td>
<td>HTTP/HTTPS/TCP/pipe/MSMQ</td>
<td>HTTP/HTTPS/TCP/pipe/messages/in-process</td>
</tr>
<tr>
<td>HTTP/HTTPS server</td>
<td><code>http.sys</code></td>
<td><code>http.sys/native (winsock)</code></td>
</tr>
<tr>
<td>Authentication</td>
<td>Windows or custom</td>
<td>Windows, ORM-based, or class-driven</td>
</tr>
<tr>
<td>Authorization</td>
<td>by attribute or config files</td>
<td>per user group, or class-driven</td>
</tr>
<tr>
<td>Threading</td>
<td>by attributes</td>
<td>at service/method level</td>
</tr>
<tr>
<td>Weight</td>
<td>middle (GC, JIT, .dll)</td>
<td>low</td>
</tr>
<tr>
<td>Speed</td>
<td>good</td>
<td>high</td>
</tr>
<tr>
<td>Extensibility</td>
<td>verbose but complete</td>
<td>customizable</td>
</tr>
<tr>
<td>Standard</td>
<td>de facto</td>
<td>KISS design (e.g. JSON, HTTP)</td>
</tr>
<tr>
<td>Source code</td>
<td>closed</td>
<td>published</td>
</tr>
<tr>
<td>License</td>
<td>proprietary</td>
<td>Open</td>
</tr>
<tr>
<td>Price</td>
<td>depends</td>
<td>Free</td>
</tr>
<tr>
<td>Support</td>
<td>official + community</td>
<td>Synopse + community</td>
</tr>
<tr>
<td>Runtime required</td>
<td>.Net framework (+ISS/WAS)</td>
<td>none (blank OS)</td>
</tr>
</tbody>
</table>

About instance life time, note that in WCF `InstanceContextMode.Single` is in fact the same as `sicShared` within `mORMot` context: only one instance is used for all incoming calls and is not recycled subsequent to the calls. Therefore, `sicSingle` mode (which is `mORMot`'s default) maps `InstanceContextMode.PerCall` in WCF, meaning that one instance is used per call.

We may be tempted to say that `mORMot` SOA architecture is almost complete, even for a young and Open Source project. Some features (like per user, per group or client-driven instance life time, or Windows Messages local communication) are even unique to `mORMot`. In fact, `sicClientDriven` is pretty convenient when implementing a Service Oriented Architecture.

Of course, WCF features its SOAP-based architecture. But WCF also suffers from it: due to this ground-up message design, it will always endure its SOAP overweight, which is "Simple" only by name, not by reputation.

If you need to communicate with an external service provider, you can easily create a SOAP gateway from Delphi, as such:
- Import the WSDL (Web Service Definition Language) definition of a web service and turn it into a Delphi import unit;
- Publish the interface as a `mORMot` server-side implementation class.

Since SOAP features a lot of requirements, and expects some plumping according to its format
(especially when services are provided from C# or Java), we choose to not re-invent the wheel this time, and rely on existing Delphi libraries (available within the Delphi IDE) for this purpose. If you need a cross-platform SOAP 1.1 compatible solution, or if you version of Delphi does not include SOAP process, you may take a look at http://wiki.freepascal.org/Web_Service_Toolkit., which is a web services package for FPC, Lazarus and Delphi.

But for service communication within the mORMot application domain, the RESTful / JSON approach gives much better performance and ease of use. You do not have to play with WSDL or unit wrappers, just share some interface definition between clients and servers. Once you have used the ServiceRegister() or ServiceDefine() methods of mORMot, you will find out how the WCF plumbing is over-sized and over-complicated: imagine that WCF allows only one end-point per interface/contract - in a SOLID design principles (page 389) world, where interface segregation should reign, it is not the easier way to go!

Optionally, mORMot's interface based services allow to publish their result as XML, and encode the incoming parameters at URI level. It makes it a good alternative to SOAP, in the XML world.

At this time, the only missing feature of mORMot's SOA is transactional process, which must be handled on server side, within the service implementation (e.g. with explicit commit or rollback).
17. Cross-Platform clients

Current version of the main framework units target only Win32 / Win64 systems under Delphi, and (in a preliminary state) Windows or Linux under FPC.

It allows to make easy self-hosting of mORMot servers for local business applications in any corporation, or pay cheap hosting in the Cloud, since mORMot CPU and RAM expectations are much lower than a regular IIS-WCF-MSSQL-.Net stack.

But in a Service-Oriented Architecture (SOA) (page 90), you will probably need to create clients for platforms outside the support platform sets world, especially mobile devices or AJAX applications.

A set of cross-platform client units is therefore available in the CrossPlatform sub-folder of the source code repository. It allows writing any client in modern object pascal language, for:
- Any version of Delphi, on any platform (Mac OSX, or any mobile supported devices);
- FreePascal Compiler (in 2.6.4, 2.7.1 or 3.x branches - preferred is 3.2 fixes);
- Smart Mobile Studio (2.1 and up), to create AJAX or mobile applications (via PhoneGap, if needed).

The units are the following:

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCrossPlatformREST.pas</td>
<td>Main unit, implementing secured ORM and SOA RESTful client</td>
</tr>
<tr>
<td>SynCrossPlatformCrypto.pas</td>
<td>SHA256 and crc32 algorithms, used for authentication</td>
</tr>
<tr>
<td>SynCrossPlatformJSON.pas</td>
<td>Optimized JSON process (not used by Smart)</td>
</tr>
<tr>
<td>SynCrossPlatformSpecific.pas</td>
<td>System-specific functions, e.g. HTTP clients</td>
</tr>
</tbody>
</table>
This set of units will provide a solid and shared ground for the any kind of clients:
- Connection to a mORMot server via HTTP, with full REST support;
- Support of weak or default authentication to secure the transfer - see below (page 546);
- Definition of the TSQLRecord class, using RTTI when available on Delphi or FreePascal, and generated code for Smart Mobile Studio;
- Remote CRUD operations, via JSON and REST, with a TSQLRestClientURI class, with the same methods as with the mORMot.pas framework unit;
- Optimized TSQLTableJSON class to handle a JSON result table, as returned by mORMot's REST server ORM - see JSON (not) expanded layouts (page 316);
- Batch process - see BATCH sequences for adding/updating/deleting records (page 350) - for transactional and high-speed writes;
- Client-Server services via methods (page 373) with parameters marshalling;
- Client-Server services via interfaces (page 419) with parameters marshalling and instance-life time;
- Mapping of most supported field types, including e.g. ISO 8601 date/time encoding, BLOBs and TModTime/TCreateTime - see TSQLRecord fields definition (page 131);
- Complex record types are also exported and consumed via JSON, on all platforms (for both ORM and SOA methods);
- Integrated debugging methods, used by both ORM and SOA process, able to log into a local file or to a remote server - see below (page 639);
- Some cross-platform low-level functions and types definitions, to help share as much code as possible between your projects.

In the future, C# or Java clients may be written. The CrossPlatform sub-folder code could be used as reference, to write minimal and efficient clients on any platform. Our REST model is pretty straightforward and standard, and use of JSON tends to leverage a lot of potential marshalling issues which may occur with XML or binary formats.

In practice, a code generator embedded in the mORMot server can be used to create the client wrappers, using the below (page 505) included on the server side. With a click, you can generate and download a client source file for any supported platform. A set of .mustache templates is available, and can be customized or extended to support any new platform: any help is welcome, especially for targeting Java or C# clients.
17.1. Available client platforms

17.1.1. Delphi FMX / FreePascal FCL cross-platform support

Latest versions of Delphi include the FireMonkey (FMX) framework, able to deliver multi-device, true native applications for Windows, Mac OSX, Android and iOS (iPhone/iPad).

Our SynCrossPlatform* units are able to easily create clients for those platforms.

Similarly, these units can be compiled with FreePascal, so that any mORMot server could be consumed from the numerous supported platforms of this compiler.

In order to use those units, ensure in your IDE that the CrossPlatform sub-folder of the mORMot source code repository is defined in your Library Search Path.

17.1.1.1. Cross-platform JSON

We developed our own cross-platform JSON process unit in SynCrossPlatformJSON.pas, shared with Delphi and FreePascal.

In fact, it appears to be easier to use (since it is variant-based and with late-binding abilities) and run much faster than the official DBXJSON.pas unit shipped with latest versions of Delphi, as stated by the "25 - JSON performance" sample:

<table>
<thead>
<tr>
<th></th>
<th>Assertions Passed</th>
<th>Time (ms)</th>
<th>Throughput (tps)</th>
<th>Memory Consumption (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopse crossplatform</td>
<td>41,135</td>
<td>20.56</td>
<td>400,048</td>
<td>1.9</td>
</tr>
<tr>
<td>DBXJSON</td>
<td>41,136</td>
<td>240.84</td>
<td>34,159</td>
<td>9.9</td>
</tr>
</tbody>
</table>

Our TSQLTableJSON class is more than 10 times faster than standard DBXJSON unit, when processing a list of results as returned by a mORMot server.

The latest value on each line above is the memory consumption. It should be of high interest on mobile platforms, where memory allocation tends to be much slower and sensitive than on Windows (where the FastMM4 memory manager does wonders). Our unit consumes 5 times less memory than the RTL's version.

We did not include XSuperObject here, which is cross-platform, but performs even worse than DBXJSON in terms of speed. Other libraries - as SuperObject or dwsJSON - are not cross-platform. See http://blog.synopse.info/post/json-benchmark-delphi-mormot-superobject-dwsjson-dbxjson.. for details about this comparison.

A special mention is due to dwsJSON, which performs very well, but only on Windows, and is slower than mORMot's implementation:

<table>
<thead>
<tr>
<th></th>
<th>Assertions Passed</th>
<th>Time (ms)</th>
<th>Throughput (tps)</th>
<th>Memory Consumption (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopse ORM loop</td>
<td>41,135</td>
<td>6.18</td>
<td>1,330,153</td>
<td>1.1</td>
</tr>
<tr>
<td>Synopse ORM list</td>
<td>41,135</td>
<td>6.47</td>
<td>1,270,775</td>
<td>952 KB</td>
</tr>
<tr>
<td>DBXJSON</td>
<td>41,136</td>
<td>20.56</td>
<td>400,048</td>
<td>1.9</td>
</tr>
<tr>
<td>Super object properties</td>
<td>41,136</td>
<td>2.20</td>
<td>3,739</td>
<td>6.3</td>
</tr>
<tr>
<td>dwsJSON</td>
<td>41,136</td>
<td>32.05</td>
<td>256,628</td>
<td>4.7</td>
</tr>
<tr>
<td>DBXJSON</td>
<td>41,136</td>
<td>240.84</td>
<td>34,159</td>
<td>9.9</td>
</tr>
</tbody>
</table>

The "Synopse ORM" lines stand for the TSQLTableJSON class as implemented in mORMot.pas. It uses our optimized UTF-8 functions and classes, in-place escaping together with our RawUTF8 custom string type as implemented in SynCommons.pas, so that it is 3 times faster than our cross-platform units, and 40 times than DBXJSON, using much less memory. Some tricks used by Synopse ORM rely on pointers and are not compatible with the NextGen compiler or the official Delphi road-map, so the Synopse crossplatform uses diverse algorithm, but offers still pretty good performance.

This unit features a TJSONVariantData custom variant type, similar to TDocVariant custom variant.
type (page 112), available in the main mORMot framework.
It allows writing such nice and readable code, with late-binding:

```pascal
var
doc: variant;
json, json2: string;
...
doc := JSONVariant('"test":1234,"name":"Joh\n\n\ nr","zero":0.0');
assert(doc.test=1234);
assert(doc.name='Joh\n\n\ nr');
assert(doc.name2=null);
assert(doc.zero=0);
json := doc;  // conversion to JSON text when assigned to a string variable
assert(json='"test":1234,"name":"Joh\n\n\ nr","zero":0.0');
doc.name2 := 3.1415926;
doc.name := 'John';
json := doc;
assert(json='"test":1234,"name":"John","zero":0,"name2":3.1415926'});
```

The unit is also able to serialize any TPersistent class, i.e. all published properties could be written or read from a JSON object representation. It also handles nested objects, stored as TCollection.

See for instance in the SynCrossPlatformTests unit:

```pascal
type
TMainNested = class(TCollectionItem)
private
  fNumber: double;
fIdent: RawUTF8;
published
  property Ident: RawUTF8 read fIdent write fIdent;
  property Number: double read fNumber write fNumber;
end;

TMain = class(TPersistent)
private
  fName: RawUTF8;
fNested: TCollection;
  fList: TStringList;
public
  constructor Create;
destructor Destroy; override;
published
  property Name: RawUTF8 read fName write fName;
  property Nested: TCollection read fNested;
  property List: TStringList read fList;
end;

obj1 := TMain.Create;
obj2 := TMain.Create;
....
obj1.Name := IntToStr(i);
item := obj1.Nested.Add as TMainNested;
item.Ident := obj1.Name;
item.Number := i/2;
obj1.list.Add(obj1.Name);
json := ObjectToJSON(obj1);
if i=1 then
  assert(json='"Name":"1","Nested":["Ident":"1","Number":0.5],"List":["1"]'});
JSONToObject(obj2, json);
assert(obj2.Nested.Count=1);
json2 := ObjectToJSON(obj2);
assert(json2=json);
...
```

Of course, this serialization feature is used for the TSQLRecord ORM class.

Due to lack of RTTI, record serialization is supported via some functions generated by the server with
the code wrappers.

17.1.1.2. Delphi OSX and NextGen

In order to be compliant with the NextGen revision, our SynCrossPlatform* units follow the expectations of this new family of cross-compilers, which targets Android and iOS. In particular, we rely only on the string type for text process and storage, even at JSON level, and we tried to make object allocation ARC-compatible. Some types have been defined, e.g. THttpBody, TUTF8Buffer or AnsiChar, to ensure that our units will compile on all supported platforms.

On Delphi, the Indy library is used for HTTP requests. It is cross-platform by nature, so should work on any supported system. For SSL support with iOS and Android clients, please follow instructions at http://blog.marcocantu.com/blog/using_ssl_delphi_ios.html.. you may also download the needed libcrypto.a and libssl.a files from http://indy.fulgan.com/SSL/OpenSSLStaticLibs.7z.

Feedback is needed for the mobile targets, via FMX. In fact, we rely for our own projects on Smart Mobile Studio for our mobile applications, so the Synopse team did not test Delphi NextGen platforms (i.e. iOS and Android) as deep as other systems. Your input will be very valuable and welcome, here!

17.1.1.3. FreePascal clients

SynCrossPlatform* units support the FreePascal Compiler, in its 2.7.1 / 3.x branches. Most of the code is shared with Delphi, including RTTI support and all supported types.

Some restrictions apply, though.

Due to a bug in FreePascal implementation of variant late binding, the following code won't work as expected on older revisions of FPC:

```pascal
doc.name2 := 3.1415926;
doc.name := 'John';
```

Under oldest FreePascal, you have to write:

```pascal
TJSONVariantData(doc)['name2'] := 3.1415926;
TJSONVariantData(doc)['name'] := 'John';
```

In fact, the way late-binding properties are implemented in the FreePascal in some fully compatible with Delphi expectations. The FreePascal maintainers did some initial fix (the variant instance is now passed by reference), so above code seems to work on current FPC trunk.

As a result, direct access to TJSONVariantData instances, and not a variant variable, may be both safer and faster when using FPC.

In the Lazarus IDE, we also observed that the debugger is not able to handle our custom variant type. If you look at any TJSONVariantData instance with the debugger, an error message "unsupported variant type" will appear. As far as we found out, this is a Lazarus limitation. Delphi, on its side, is able to display any custom variant type in its debugger, after conversion to string, i.e. its JSON representation.

Another issue with the 2.7.1 / 3.1.1 revisions is how the new string type is implemented. In fact, if you use a string variable containing an UTF-8 encoded text, then the following line will reset the result code page to the system code page:

```pascal
function StringToJSON(const Text: string): string;
```

```pascal
... result := ''+copy(Text,1,j-1); // here FPC 2.7.1 erases UTF-8 encoding
```
It sounds like if `''` will force the code page of result to be not an UTF-8 content. With Delphi, this kind of statements work as expected, even for AnsiString values, and `''` constant is handled as RawByteString. We were not able to find an easy and safe workaround for FPC yet. Input is welcome in this area, from any expert!

You have to take care of this limitation, if you target the Windows operating system with FPC (and Lazarus). Under other systems, the default code page is likely to be UTF-8, so in this case our SynCrossPlatform* units will work as expected.

We found out the FreePascal compiler to work very well, and result in small and fast executables. For most common work, timing is comparable with Delphi. The memory manager is less optimized than FastMM4 for rough simple threaded tests, but is cross-platform and designed to be more efficient in multi-thread mode: in fact, it has no giant lock, as FastMM4 suffers.

### 17.1.1.4. Local or remote logging

You can use the TSQLRest.Log() overloaded methods to log any content into a file or a remote server.

All ORM and SOA functions of the TSQLRest instance will create the expected log, just with the main mORMot units running on Win32/Win64 - see below (page 636).

For instance, here are some log entries created during the RegressionTest.dpr process:

```
16:47:15 Trace  POST root/People status=201 state=847 in=92 out=0
16:47:15 DB  People.ID=200 created from
{"FirstName":"First200","LastName":"Last200","YearOfBirth":2000,"YearOfDeath":2025,"Sexe":0}
16:47:15 SQL  select RowID,FirstName,LastName,YearOfBirth,YearOfDeath,Sexe from People
16:47:15 Trace  GET root?sql=select+RowID%2CFirstName%2CLastName%2CYearOfBirth%2CYearOfDeath%2CSexe+from+People status=200 state=847 in=0 out=21078
16:47:15 SQL  select RowID,YearOfBirth,YearOfDeath from People
16:47:15 Trace  GET root?sql=select+RowID%2CYearOfBirth%2CYearOfDeath+from+People status=200 state=847 in=0 out=10694
16:47:15 SQL  select RowID,FirstName,LastName,YearOfBirth,YearOfDeath,Sexe from People where yearofbirth=:(1900):
16:47:15 Trace  GET root?sql=select+RowID%2CFirstName%2CLastName%2CYearOfBirth%2CYearOfDeath%2CSexe+from+People+where+ yearofbirth%3D%3A%281900%29%3A status=200 state=847 in=0 out=107
16:47:15 Trace  DELETE root/People/16 status=200 state=848 in=0 out=0
16:47:15 DB  Delete People.ID=16
```

Then, our Log View tool is able to run as a remote log server, and display the incoming events in real-time - see below (page 639).

Having such logs available will be pretty convenient, especially when debugging applications on a mobile device, or a remote computer.

### 17.1.2. Smart Mobile Studio support

Smart Mobile Studio - see [http://www.smartmobilestudio.com](http://www.smartmobilestudio.com) - is a complete RAD environment for writing cutting edge HTML5 mobile applications. It ships with a fully fledged compiler capable of compiling Object Pascal (in a modern dialect call SmartPascal) into highly optimized and raw JavaScript.

There are several solutions able to compile to JavaScript. In fact, we can find several families of compilers:

- **JavaScript** super-sets, adding optional strong typing, and classes, close to the ECMAScript Sixth Edition standard.
Edition: the current main language in this category is certainly TypeScript, designed by Anders Hejlsberg (father of both the Delphi language and C#), and published by Microsoft;
- New languages, dedicated to make writing JavaScript programs easier, with an alternative syntax and new concepts (like classes, lambdas, scoping, splats, comprehensions...): most relevant languages of this family are CoffeeScript and Dart;
- High-level languages, like Google Web Toolkit (compiling Java code), JSIL (from C# via Mono), or Smart Mobile Studio (from object pascal);
- Low-level languages, like Emscripten (compiling C/C++ from LLVM byte-code, using asm.js).

Of course, from our point of view, use of modern object pascal is of great interest, since it will leverage our own coding skills, and make us able to share code between client and server sides.

17.1.2.1. Beyond JavaScript

The Smart Pascal language brings strong typing, true OOP to JavaScript, including classes, partial classes, interfaces, inheritance, polymorphism, virtual and abstract classes and methods, helpers, closures, lambdas, enumerations and sets, getter/setter expressions, operator overloading, contract programming. But you can still unleash the power of JavaScript (some may say "the good parts"), if needed: the variant type is used to allow dynamic typing, and you can write some JavaScript code as an asm .. end block.

See http://en.wikipedia.org/wiki/The_Smart_Pascal_programming_language..

The resulting HTML5 project is self-sufficient with no external JavaScript library, and is compiled as a single index.html file (including its css, if needed). The JavaScript code generated by the compiler (written in Delphi by Eric Grange), is of very high quality, optimized for best execution performance (either in JIT or V8), has low memory consumption, and can be compressed and/or obfuscated.

The SmartCL runtime library encapsulate HTML5 APIs in a set of pure pascal classes and functions, and an IDE with an integrated form designer is available. You can debug your application directly within the IDE (since revision 2.1 - even if it is not yet always stable) or within your browser (IE, Chrome or FireBug have great debuggers), with step-by-step execution of the object pascal code (if you define "Add source map (for debugging)" in Project Options / Linker).

Using a third-party tool like PhoneGap - see http://phonegap.com.. - you will be able to supply your customers with true native iOS or Android applications, running without any network, and accessing the full power of any modern Smart Phone. Resulting applications will be much smaller in size than the one generated by Delphi FMX (a simple Smart RESTful client with a login form and ORM + SOA tests is zipped as 40 KB), and will work seamlessly on all HTML5 platforms, including most mobile (like Windows Phone, Blackberry, Firefox OS, or webOS) or desktop (Windows, Linux, BSD, MacOS) architectures.

Smart Mobile Studio is therefore a great platform for implementing rich client-side AJAX or Mobile applications, to work with our client-server mORMot framework.

17.1.2.2. Using Smart Mobile Studio with mORMot

There is no package to be installed within the Smart Mobile Studio IDE. The client units will be generated directly from the mORMot server.

Any edition of Smart - see http://smartmobilestudio.com/feature-matrix.. - is enough: you do not need to pay for the Enterprise edition to consume mORMot services. But of course, the Professionnal edition is recommended, since the Basic edition does not allow to create forms from the IDE, which is the main point for an AJAX application.
In contrast to the wrappers available in the *Professional* edition of Smart, for accessing *RemObjects* or *DataSnap* servers, our *mORMot* clients are 100% written in the *SmartPascal* dialect. There is no need to link an external .js library to your executable, and you will benefit of the obfuscation and smart linking features of the Smart compiler.

The only requirement is to copy the *mORMot* cross-platform units to your *Smart Mobile Studio* installation. This can be done in three copy instructions:

```plaintext
xcopy SynCrossPlatformSpecific.pas "c:\ProgramData\Optimale Systemer AS\Smart Mobile Studio\Libraries" /Y
xcopy SynCrossPlatformCrypto.pas "c:\ProgramData\Optimale Systemer AS\Smart Mobile Studio\Libraries" /Y
xcopy SynCrossPlatformREST.pas "c:\ProgramData\Optimale Systemer AS\Smart Mobile Studio\Libraries" /Y
```

You can find a corresponding BATCH file in the CrossPlatform folder, and in SQLite3\Samples\29 - SmartMobileStudio Client\CopySynCrossPlatformUnits.bat.

In fact, the SynCrossPlatformJSON.pas unit is not used under *Smart Mobile Studio*: we use the built-in JSON serialization features of *JavaScript*, using variant dynamic type, and the standard JSON.Stringify() and JSON.Parse() functions.

### 17.1.3. Remote logging

Since there is no true file system API available under a HTML5 sand-boxed application, logging to a local file is not an option. Even when packaged with *PhoneGap*, local log files are not convenient to work with.

Generated logs will have the same methods and format as with *Delphi* or *FreePascal*—see *Local or remote logging* (page 486). TSQLRest.Log(E: Exception) method will also log the stack trace of the exception! Our LogView tool - see below (page 639) - is able to run as a simple but efficient remote log server and viewer, shared with regular or cross-platform units of the framework.

A dedicated asynchronous implementation has been refined for *Smart Mobile Studio* clients, so that several events will be gathered and sent at once to the remote server, to maximize bandwidth use and let the application be still responsive.

It allows even complex mobile applications to be debugged with ease, on any device, even over WiFi or 3G/4G networks. Your support could ask your customer to enable logging for a particular case, then see in real time what is wrong with your application.
17.2. Generating client wrappers

Even if it is feasible to write the client code by hand, your mORMot server is able to create the source code needed for client access, via a dedicated method-based service, and a set of Mustache-based templates - see below (page 505).

The following templates are available in the CrossPlatform\templates folder:

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Compiler Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>CrossPlatform.pas.mustache</td>
<td>Delphi / FPC SynCrossPlatform* units</td>
</tr>
<tr>
<td>Delphi.pas.mustache</td>
<td>Delphi Win32/Win64 mORMot units</td>
</tr>
<tr>
<td>SmartMobileStudio.pas.mustache</td>
<td>Smart Mobile Studio 2.1</td>
</tr>
</tbody>
</table>

In the future, other wrappers may be added. And you can write your own, which could be included within the framework source! Your input is warmly welcome, especially if you want to write a template for Java or C# client. The generated data context already contains the data types corresponding to those compilers: e.g. a mORMot's RawUTF8 field or parameter could be identified as "typeCS":"string" or "typeJava":"String" in addition to "typeDelphi":"RawUTF8" and "typePascal":"string".

17.2.1. Publishing the code generator

By default, and for security reasons, the code generation is not embedded to your mORMot RESTful server. In fact, the mORMotWrapper.pas unit will link both mORMot.pas and SynMustache.pas units, and use Mustache templates to generate code for a given TSQLRestServer instance.

We will start from the interface-based service Sample code (page 437) as defined in the "SQLite3\Samples\14 - Interface based services" folder. After some minor modifications, we copied the server source code into "SQLite3\Samples\27 - CrossPlatform Clients\Project14ServerHttpWrapper.dpr":

```
program Project14ServerHttpWrapper;

{$APP TYPE CONSOLE}

uses
  SysUtils,
  Classes,
  SynCommons,
  mORMot,
  mORMotHttpServer,
  mORMotWrappers,
  Project14Interface in '..\14 - Interface based services\Project14Interface.pas';

type
  TServiceCalculator = class(TInterfacedObject, ICalculator)
  public
    function Add(n1, n2: integer): integer;
  end;

function TServiceCalculator.Add(n1, n2: integer): integer;
begin
  result := n1+n2;
end;
```
var
  aModel: TSQLModel;
aServer: TSQLRestServer;
aHTTPServer: TSQLHttpServer;
begin
  // create a Data Model
  aModel := TSQLModel.Create([],ROOT_NAME);
  try
    // initialize a TObjectList-based database engine
    aServer := TSQLRestServerFullMemory.Create(aModel,'test.json',false,true);
    try
      // add the http://localhost:888/root/wrapper code generation web page
      AddToServerWrapperMethod(aServer,
        ['...\..\..\..\..<CrossPlatform\templates>','...\..\..\..\..<CrossPlatform\templates>]);
      // register our ICalculator service on the server side
      aServer.ServiceDefine(TServiceCalculator,[ICalculator],sicShared);
      // launch the HTTP server
      aHTTPServer := TSQLHttpServer.Create(PORT_NAME,[aServer],'+',useHttpApiRegisteringURI);
      try
        aHTTPServer.AccessControlAllowOrigin := '*'; // for AJAX requests to work
        writeln('#10'Background server is running.');
        writeln('You can test http://localhost:',PORT_NAME,'/wrapper');
        writeln('#10'Press [Enter] to close the server.'#10);
        readln;
      finally
        aHTTPServer.Free;
      end;
      finally
        aServer.Free;
      end;
      finally
        aModel.Free;
    end;
  finally
    aServer.Free;
  end;
end.

As you can see, we just added a reference to the mORMotWrappers unit, and a call to AddToServerWrapperMethod() in order to publish the available code generators.

Now, if you run the Project14ServerHttpWrapper server, and point your favorite browser to http://localhost:888/root/wrapper you will see the following page:

**Client Wrappers**

**Available Templates:**

* CrossPlatform
  *mORMotClient.pas - download as file - see as text - see template*

* Delphi
  *mORMotClient.pas - download as file - see as text - see template*

* SmartMobileStudio
  *mORMotClient.pas - download as file - see as text - see template*

You can also retrieve the corresponding template context.

Each of the *.mustache template available in the specified folder is listed here. Links above will allow downloading a client source code unit, or displaying it as text in the browser. The template can also be displayed un-rendered, for reference. As true Mustache templates, the source code files are
generated from a data context, which can be displayed, as JSON, from the "template context" link. It may help you when debugging your own templates. Note that if you modify and save a .mustache template file, just re-load the "see as text" browser page and your modification is taken in account immediately (you do not need to restart the server).

Generated source code will follow the template name, and here will always be downloaded as mORMotClient.pas. Of course, you can change the unit name for your end-user application. It could be even mandatory if the same client will access to several mORMot servers at once, which could be the case in a Service-Oriented Architecture (SOA) (page 90) project.

Just ensure that you will never change the mORMotClient.pas generated content by hand. If necessary, you can create and customize your own Mustache template, to be used for your exact purpose. By design, such automated code generation will require to re-create the client unit each time the server ORM or SOA structure is modified. In fact, as stated in the mORMotClient.pas comment, any manual modification of this file may be lost after regeneration. You have been warned!

For publishing the wrappers for a REST / ORM oriented program, take a look at the '28 - Simple RESTful ORM Server' sample.

If you feel that the current templates have some issues or need some enhancements, you are very welcome to send us your change requests on our forums. Once you are used at it, Mustache templates are fairly easy to work with. Similarly, if you find out that some information is missing in the generated data context, e.g. for a new platform or language, we will be pleased to enhance the official mORMotWrapper.pas process.

17.2.2. Delphi / FreePascal client samples

The "27 - CrossPlatform Clients.dpr" sample creates a mORMot server with its own ORM data model, containing a TSQLRecordPeople class, and a set of interface-based SOA services, some including complex types like a record.

Then this sample uses a generated mORMotClient.pas, retrieved from the "download as file" link of the CrossPlatform template above. Its set of regression tests (written using a small cross-platform TSynTest unit test class) will then perform remote ORM and SOA calls to the PeopleServer embedded instance, over all supported authentication schemes - see below (page 546):

Cross Platform Units for mORMot
---------------------------------

1. Running "Iso8601DateTime"
   30003 tests passed in 00:00:018
2. Running "Base64Encoding"
   304 tests passed in 00:00:000
3. Running "JSON"
   18628 tests passed in 00:00:056
4. Running "Model"
   1013 tests passed in 00:00:003
5. Running "Cryptography"
   4 tests passed in 00:00:000

Tests failed: 0 / 49952
Time elapsed: 00:00:080

Cross Platform Client for mORMot without authentication
---------------------------------------------------------
1. Running "Connection"
   2 tests passed in 00:00:010
2. Running "ORM"
   4549 tests passed in 00:00:160
3. Running "ORMBatch"
   4564 tests passed in 00:00:097
4. Running "Services"
   26253 tests passed in 00:00:302
5. Running "CleanUp"
   1 tests passed in 00:00:000

Tests failed: 0 / 35369
Time elapsed: 00:00:574

Cross Platform Client for mORMot using TSQLRestServerAuthenticationNone
... 
Cross Platform Client for mORMot using TSQLRestServerAuthenticationDefault
... 

The generated mORMotClient.pas unit is used for all "Cross Platform Client" tests above, covering both ORM and SOA features of the framework.

17.2.2.1. Connection to the server
You could manually connect to a mORMot server as such:

```pascal
var Model: TSQLModel;
    Client: TSQLRestClientHTTP;
...
Model := TSQLModel.Create([TSQLAuthUser,TSQLAuthGroup,TSQLRecordPeople]);
Client := TSQLRestClientHTTP.Create('localhost',SERVER_PORT,Model);
if not Client.Connect then raise Exception.Create('Impossible to connect to the server');
if Client.ServerTimestamp=0 then raise Exception.Create('Incorrect server');
if not Client.SetUser(TSQLRestAuthenticationDefault,'User','synopse') then raise Exception.Create('Impossible to authenticate to the server');
...
```

Or you may use the GetClient() function generated in mORMotClient.pas:

```pascal
function GetClient(const aServerAddress, aUserName,aPassword: string;
aServerPort: integer=SERVER_PORT): TSQLRestClientHTTP;
Which could be used as such:
```
```pascal
var Client: TSQLRestClientHTTP;
...
Client := GetClient('localhost','User','synopse')
```

The data model and the expected authentication scheme were included in the GetClient() function, which will raise the expected ERestException in case of any connection or authentication issue.

17.2.2.2. CRUD/ORM remote access
Thanks to SynCrossPlatform* units, you could easily perform any remote ORM operation on your mORMot server, with the usual TSQLRest CRUD methods.
For instance, the RegressionTests.dpr sample performs the following operations

```pascal
fClient.CallBackGet('DropTable',[],Call,TSQLRecordPeople); // call of method-based service
check(Call.OutStatus=HTTP_SUCCESS);
people := TSQLRecordPeople.Create; // create a record ORM
try
  for i := 1 to 200 do begin
    people.FirstName := 'First'+IntToStr(i);
    people.LastName := 'Last'+IntToStr(i);
    people.YearOfBirth := i+1800;
    people.YearOfDeath := i+1825;
    people.Sexe := TPeople.Sexe(i and 1);
    check(Client.Add(people,true)=i); // add one record
  end;
finally
  people.Free;
end;
...
people := TSQLRecordPeople.CreateAndFillPrepare(fClient,'',
  'yearofbirth=?',[1900]); // parameterized query returning one or several rows
try
  n := 0;
  while people.FillOne do begin
    inc(n);
    check(people.ID=100);
    check(people.FirstName='First100');
    check(people.LastName='Last100');
    check(people.YearOfBirth=1900);
    check(people.YearOfDeath=1925);
  end;
  check(n=1); // we expected only one record here
finally
  people.Free;
end;
for i := 1 to 200 do
  if i and 15=0 then
    fClient.Delete(TSQLRecordPeople,i) else // record deletion
  if i mod 82=0 then begin
    people := TSQLRecordPeople.Create;
    try
      id := i+1;
      people.ID := id;
      people.YearOfBirth := id+1800;
      people.YearOfDeath := id+1825;
      check(fClient.Update(people,'YearOFBIRTH,YearOfDeath')); // record modification
    finally
      people.Free;
    end;
  end;
for i := 1 to 200 do begin
  people := TSQLRecordPeople.Create(fClient,i); // retrieve one instance from ID
  try
    if i and 15=0 then // was deleted
      Check(people.ID=0) else begin
        if i mod 82=0 then
          id := i+1 else // was modified
            id := i;
        Check(people.ID=id);
        Check(people.FirstName='First'+IntToStr(i));
        Check(people.LastName='Last'+IntToStr(i));
        Check(people.YearOfBirth=id+1800);
        Check(people.YearOfDeath=id+1825);
        Check(ord(people.Sexe)=i and 1);
      end;
    finally
      people.Free;
    end;
end;
```

As we already stated, BATCH mode is also supported, with the classic mORMot syntax:

```pascal
... res: TIntegerDynArray;
...
fClient.BatchStart(TSQLRecordPeople);
people := TSQLRecordPeople.Create;
try
  for i := 1 to 200 do begin
    people.FirstName := 'First'+IntToStr(i);
    people.LastName := 'Last'+IntToStr(i);
    people.YearOfBirth := i+1800;
    people.YearOfDeath := i+1825;
    fClient.BatchAdd(people,true);
  end;
finally
  people.Free;
end;
```

You can note that all above code has exactly the same structure and methods than standard mORMot clients.

The generated mORMotClient.pas unit contains all needed TSQLRecord types, and its used properties, including enumerations or complex records. The only dependency of this unit are SynCrossPlatform* units, so will be perfectly cross-platform (whereas our main SynCommons.pas and mORMot.pas units do target only Win32 and Win64).

As a result, you are able to share server and client code between a Windows project and any supported platform, even AJAX (see "Smart Mobile Studio client samples" below). A shared unique code base will eventually reduce both implementation and debugging time, which is essential to unleash your business code potential and maximize your ROI.

17.2.2.3. Service consumption

The ultimate goal of the mORMot framework is to publish your business via a Service-Oriented Architecture (SOA) (page 90).

As a consequence, those services should be made available from any kind of device or platform, even outside the Windows world. The server is able to generate client wrappers code, which could be used to consume any Client-Server services via interfaces (page 419) using any supported authentication scheme - see below (page 546).

Here is an extract of the mORMotClient.pas unit as generated for the RegressionTests.dpr sample:

```pascal
type
  /// service implemented by TServiceCalculator
  /// - you can access this service as such:
```
As you can see, a dedicated class has been generated to consume the server-side ICalculator interface-based service, in its own ICalculator client-side type. It is able to handle complex types, like enumerations (e.g. TPeopleSexe) and records (e.g. TTestCustomJSONArraySimpleArray), which are also defined in the very same mORMotClient.pas unit.

You can note that the RawUTF8 type has been changed into the standard Delphi / FreePascal string type, since it is the native type used by our SynCrossPlatformJSON.pas unit for all its JSON marshalling. Of course, under latest version of Delphi and FreePascal, this kind of content may be Unicode encoded (either as UTF-16 for the string = UnicodeString Delphi type, or as UTF-8 for the FreePascal / Lazarus string type).

The supplied regression tests show how to use remotely those services:

```pascal
var calc: ICalculator;
    i, j: integer;
    sex: TPeopleSexe;
    name: string;

    calc := TServiceCalculator.Create(fClient);
    check(calc.InstanceImplementation=sicShared);
    check(calc.ServiceName='Calculator');
    for i := 1 to 200 do
        check(calc.Add(i,i+1)=i*2+1);
    for i := 1 to 200 do begin
        sex := TPeopleSexe(i and 1);
        name := 'Smith';
        calc.ToText(i,'$',sex,name);
        check(sex=sFemale);
        check(name=format('$ %d for %s Smith', [i, SEX_TEXT[i and 1]]));
    end;
```

As with regular mORMot client code, a TServiceCalculator instance is created and is assigned to an ICalculator local variable. As such, no try ... finally Calc.Free end block is mandatory here, to avoid any memory leak: the compiler will create such an hidden block for the Calc: ICalculator variable scope.

The service-side contract of the ICalculator signature is retrieved and checked within TServiceCalculator.Create, and will raise an ERestException if it does not match the contract
identified in mORMotClient.pas.

The cross-platform clients are able to manage the service instance life-time, especially the sicPerClient mode. In this case, an implementation class instance will be created on the server for each client, until the corresponding interface instance will released (i.e. out of scope or assigned to nil), which will release the server-side instance - just like with a regular mORMot client code.

Note that all process here is executed synchronously, i.e. in blocking mode. It is up to you to ensure that your application is able to still be responsive, even if the server does a lot of process, so may be late to answer. A dedicated thread may help in this case.

### 17.2.3. Smart Mobile Studio client samples

In addition to Delphi and FreePascal clients, our framework is able to access any mORMot server from HTML5 / AJAX rich client, thanks to Smart Mobile Studio.

#### 17.2.3.1. Adding two numbers in AJAX

You can find in SQLite3\Samples\27 - CrossPlatform Clients\SmartMobileStudio a simple client for the TServiceCalculator.Add() interface-based service. If your Project14ServerHttpWrapper server is running, you can just point to the supplied www\index.html file in the sub-folder. You will then see a web page with a "Server Connect" button, and if you click on it, you will be able to add two numbers. This a full HTML5 web application, connecting securely to your mORMot server, which will work from any desktop browser (on Windows, Mac OS X, or Linux), or from any mobile device (either iPhone / iPad / Android / Windows 8 Mobile).

In order to create the application, we just clicked on "download as file" in the SmartMobileStudio link in the web page, and copied the generated file in the source folder of a new Smart Mobile project. Of course, we did copy the needed SynCrossPlatform*.pas units from the mORMot source code tree into the Smart library folder, as stated above. Just ensure you run CopySynCrossPlatformUnits.bat from the CrossPlatform folder at least once from the latest revision of the framework source code.

Then, on the form visual editor, we added a BtnConnect button, then a PanelCompute panel with two edit fields named EditA and EditB, and two other buttons, named BtnComputeAsynch and BtnComputeSynch. A LabelResult label will be used to display the computation result. The BtnConnect is a toggle which will show or display the PanelCompute panel, which is hidden by default, depending on the connection status.
In the Form1.pas unit source code side, we added a reference to our both SynCrossPlatformREST and mORMotClient units, and some events to the buttons:

```pascal
unit Form1;

interface

uses
  SmartCL.System, SmartCL.Graphics, SmartCL.Components, SmartCL.Forms,
  SmartCL.Fonts, SmartCL.Borders, SmartCL.Application, SmartCL.Controls.Panel,
  SmartCL.Controls.Label, SmartCL.Controls.EditBox, SmartCL.Controls.Button,
  SynCrossPlatformREST, mORMotClient;

type
  TForm1 = class(TW3Form)
  private
    {$I 'Form1:intf'}
    Client: TSQLRestClientURI;
    procedure GetClient(const aServerAddress, aUserName, aPassword: string;
                         onSuccess, onError: TSQLRestEvent; aServerPort: integer=SERVER_PORT);
  protected
    procedure BtnComputeSynchClick(Sender: TObject);
    procedure BtnComputeAsynchClick(Sender: TObject);
    procedure BtnConnectClick(Sender: TObject);
  public
    procedure InitializeForm; override;
    procedure InitializeObject; override;
    procedure Resize; override;
  end;

{$I 'Form1:intf'}

The BtnConnect event will connect asynchronously to the server, using 'User' as log-on name, and 'synopse' as password (those as the framework defaults).

We just use the GetClient() function, as published in our generated mORMotClient.pas unit:

```pascal
procedure TForm1.BtnConnectClick(Sender: TObject);

procedure TForm1.BtnComputeSynchClick(Sender: TObject);
procedure TForm1.BtnComputeAsynchClick(Sender: TObject);
```
begin
if Client=nil then
  GetClient('127.0.0.1','User','synopse',
    lambda (aClient: TSQLRestClientURI)
    PanelCompute.Visible := true;
    W3Label1.Visible := true;
    W3Label2.Visible := true;
    LabelConnect.Caption := '';
    BtnConnect.Caption := 'Disconnect';
    LabelResult.Caption := '';
    Client := aClient;
  end,
  lambda
    ShowMessage('Impossible to connect to the server!');
end
else begin
  PanelCompute.Visible := false;
  BtnConnect.Caption := 'Server Connect';
  Client.Free;
  Client := nil;
end;
end;

The GetClient() function expects two callbacks, respectively onSuccess and onError, which are implemented here with two SmartPascal lambda blocks.

Now that we are connected to the server, let’s do some useful computation!
As you can see in the mORMotClient.pas generated unit, our interface-based service can be accessed via a SmartPascal TServiceCalculator class (and not an interface), with two variations of each methods: one asynchronous method - e.g. TServiceCalculator.Add() - expecting success/error callbacks, and one synchronous (blocking) method - e.g. TServiceCalculator._Add():
Of course, the synchronous code is much easier to follow and maintain. To be fair, the *SmartPascal* lambda syntax is not difficult to read nor write. In the browser debugger, you can easily set a break point within any lambda block, and debug your code.

Note that if the server is slow to answer, your whole web application will be unresponsive, and the browser may even complain about the page, proposing the kill its process! As a consequence, simple services may be written in a synchronous manner, but your serious business code should rather use asynchronous callbacks, just as with any modern AJAX application.

Thanks to the *Smart Linking* feature of its compiler, only the used version of the unit will be converted to *JavaScript* and included in the final `index.html` HTML5 file. So having both synchronous and asynchronous versions of each method at hand is not an issue.

### 17.2.3.2. CRUD/ORM remote access

If the server did have some ORM model, its `TSQLRecord` classes will also be part of the `mORMotClient.pas` generated unit. All types, even complex record structures, will be marshaled as expected.

For instance, if you run the `RegressionTestsServer.dpr` server (available in the same folder), a much more complete unit could be generated from `http://localhost:888/root/wrapper`:

```pascal
type // define some enumeration types, used below
TPeopleSexe = (sFemale, sMale);
TRecordEnum = (reOne, reTwo, reLast);

type // define some record types, used as properties below
TTestCustomJSONArraySimpleArray = record
  F: string;
  G: array of string;
  H: record
    H1: integer;
    H2: string;
    H3: record
      H3a: boolean;
      H3b: TSQLRawBlob;
    end;
  I: TDateTime;
  J: array of record
    J1: byte;
    J2: TGUID;
    J3: TRecordEnum;
end;
end;

type // service accessible via http://localhost:888/root/Calculator
// - this service will run in sicShared mode
// - synchronous and asynchronous methods are available, depending on use case
// - synchronous _^() methods will block the browser execution, so won't be
// appropriate for long process - on error, they may raise EServiceException
TServiceCalculator = class(TServiceClientAbstract)
  public
    // will initialize an access to the remote service
    constructor Create(aClient: TSQLRestClientURI); override;
    procedure Add(n1: integer; n2: integer;
      onSuccess: procedure(Result: integer); onError: TSQLRestEvent);
    function _Add(const n1: integer; const n2: integer): integer;
end;
```
In the above code, you can see several methods to the ICalculator service, some involving the complex TTestCustomJSONArraySimpleArray record type. The implementation section of the unit will in fact allow serialization of such records to/from JSON, even with obfuscated JavaScript field names.

Some enumerations types are also defined, so will help your business code be very expressive, thanks to the SmartPascal strong typing. This is a huge improvement when compared to JavaScript native weak and dynamic typing.

There is a TSQLRecordPeople class generated, which will map the following Delphi class type, as defined in the PeopleServer.pas unit:

```pascal
TSQLRecordPeople = class(TSQLRecord)
  protected
    fData: TSQLRawBlob;
    fFirstName: RawUTF8;
    fLastName: RawUTF8;
    fYearOfBirth: integer;
    fYearOfDeath: word;
    fSexe: TPeopleSexe;
    fSimple: TTestCustomJSONArraySimpleArray;
  public
    class procedure InternalRegisterCustomProperties(Props: TSQLRecordProperties); override;
end;
```
Here, a complex TTestCustomJSONArraySimpleArray record field has been published, thanks to a manual InternalRegisterCustomProperties() registration, as we already stated above. Since SmartPascal is limited in terms of RTTI, the code generator did define some ComputeRTTI(), GetProperty() and SetProperty() protected methods, which will, at runtime, perform all the properties marshalling to and from JSON.

You can see that types like RawUTF8 in the original Delphi TSQLRecord were mapped to the standard SmartPascal string type, as expected, when converted to the mORMotClient.pas generated unit.

Your AJAX client can then access to this TSQLRecordPeople content easily, via standard CRUD operations.

See the SQLite3\Samples\29 - SmartMobileStudio Client sample, for instance the following line:

```pascal
people := new TSQLRecordPeople;
for i := 1 to 200 do begin
  assert(client.Retrieve(i,people));
  assert(people.ID=i);
  assert(people.FirstName='First'+IntToStr(i));
  assert(people.LastName='Last'+IntToStr(i));
  assert(people.YearOfBirth=id+1800);
  assert(people.YearOfDeath=id+1825);
end;
```

Here, the client variable is a TSQLRestClientURI instance, as returned by the GetClient() onSuccess callback generated in mORMotClient.pas.

You have Add() Delete() Update() FillPrepare() CreateAndFillPrepare() and Batch*() methods available, ready to safely access your data from your AJAX client.

If you update your data model on the server, just re-generate your mORMotClient.pas unit from http://localhost:888/root/wrapper, then rebuild your Smart Mobile Studio project to reflect all changes made to your ORM data model, or your SOA available services.

Thanks to the SmartPascal strong typing, any breaking change of the server expectations will immediately be reported at compilation, and not at runtime, as it will with regular JavaScript clients.
The mORMot framework allows writing rich and/or web MVC applications, relying on regular ORM and SOA methods to implement its business model and its application layer, with an optional dedicated MVC model for the HTML rendering.

18.1. Model
According to the Model-View-Controller (MVC) pattern - see Model-View-Controller (page 86) - the database schema should be handled separately from the User Interface.

The TSQLModel class centralizes all TSQLRecord inherited classes used by an application, both database-related and business-logic related.

See ORM Data Model (page 171) for how to define the model of your application.

18.2. Views
The mORMot framework also features two kinds of User Interface generation, corresponding to the MVC Views:
- For Desktop clients written in Delphi, it allows creation of Ribbon-like interfaces, with full data view and navigation as visual Grids. Reporting and edition windows can be generated in an automated way. The whole User Interface is designed in code, by some constant definitions.
- For Web clients, an optimized Mustache Template engine in pure Delphi has been integrated, and allows easy creation of HTML views, with a clear MVC design.
### 18.2.1. Desktop clients

#### 18.2.1.1. RTTI

The *Delphi* language (aka Object Pascal) provided Runtime Type Information (RTTI) more than a decade ago. In short, Runtime Type Information is information about an object’s data type that is set into memory at run-time. The RTTI support in *Delphi* has been added first and foremost to allow the design-time environment to do its job, but developers can also take advantage of it to achieve certain code simplifications. Our framework makes huge use of RTTI, from the database level to the User Interface. Therefore, the resulting program has the advantages of very fast development (Rails-like), but with the robustness of strong type syntax, and the speed of one of the best compilers available.

In short, it allows the software logic to be extracted from the code itself. Here are the places where this technology was used:

- All database structures are set in the code by normal classes definition, and most of the needed SQL code is created on the fly by the framework, before calling the *SQLite3* database engine, resulting in a true Object-relational mapping (ORM) framework;
- All User Interface is generated by the code, by using some simple data structures, relying on enumerations (see next paragraph);
- Most of the text displayed on the screen does rely on RTTI, thanks to the Camel approach (see below), ready to be translated into local languages;
- All internal Event process (such as Button press) relies on enumerations RTTI;
- Options and program parameters are using RTTI for data persistence and screen display (e.g. the Settings window of your program can be created by pure code): adding an option is a matter of a few code lines.

In *Delphi*, enumeration types or *Enum* provides a way of to define a list of values. The values have no
inherent meaning, and their ordinality follows the sequence in which the identifiers are listed. These values are written once in the code, then used everywhere in the program, even for User Interface generation.

For example, some tool-bar actions can be defined with:

```pascal
type
  /// item toolbar actions
  TBabyAction = (paCreateNew, paDelete, paEdit, paQuit);

Then this TBabyAction enumerated type is used to create the User Interface ribbon of the main window, just by creating an array of set of this kind:

BarEdit: array[0..1] of set of TBabyAction = (
  [paCreateNew, paDelete, paEdit],
  [paQuit] );
```

The caption of the buttons to be displayed on the screen is then extracted by the framework using "Camel Case": the second button, defined by the paCreateNew identifier in the source code, is displayed as "Create new" on the screen, and this "Create new" is used for direct i18n of the software. For further information about "Camel Case" and its usage in Object Pascal, Java, Dot Net, Python see [http://en.wikipedia.org/wiki/CamelCase](http://en.wikipedia.org/wiki/CamelCase). 

Advantages of the RTTI can therefore be summed up:
- Software maintainability, since the whole program logic is code-based, and the User Interface is created from it. It therefore avoid RAD (Rapid Application Development) abuse, which mix the User Interface with data logic, and could lead into "write fast, try to maintain" scenarios;
- Enhanced code security, thanks to Object Pascal strong type syntax;
- Direct database access from the language object model, without the need of writing SQL or use of a MVC framework;
- User Interface coherency, since most screen are created on the fly;
- Easy i18n of the software, without additional components or systems.

### 18.2.1.2. User Interface

User Interface generation from RTTI and the integrated reporting features will be described below (page 2513), during presentation of the Main Demo application design.

In short, such complex model including User Interface auto-creation could be written as such - extracted from unit FileTables.pas:

```pascal
function CreateFileModel(Owner: TSQLRest): TSQLModel;
begin
  result := TSQLModel.Create(Owner,
  @FileTabs, length(FileTabs), sizeof(FileTabs[0]), [],
  TypeInfo(TFileAction), TypeInfo(TFileEvent));
end;
```

All needed TSQLRecord classes are declared in a constant array, and will use TFileAction / TFileEvent enumeration types to handle the User Interface activity and Business Logic.
18.2.2. Web clients

18.2.2.1. Mustache template engine

Mustache - see http://mustache.github.io - is a well-known logic-less template engine. There is plenty of Open Source implementations around (including in JavaScript, which can be very convenient for AJAX applications on client side, for instance). For mORMot, we created the first pure Delphi implementation of it, with a perfect integration with other bricks of the framework.

Generally speaking, a Template system can be used to separate output formatting specifications, which govern the appearance and location of output text and data elements, from the executable logic which prepares the data and makes decisions about what appears in the output.

Most template systems (e.g. PHP, smarty, Razor...) feature in fact a full scripting engine within the template content. It allows powerful constructs like variable assignment or conditional statements in the middle of the HTML content. It makes it easy to modify the look of an application within the template system exclusively, without having to modify any of the underlying "application logic". They do so, however, at the cost of separation, turning the templates themselves into part of the application logic.

Mustache inherits from Google's ctemplate library, and is used in many famous applications, including the "main" Google web search, or the Twitter web site. The Mustache template system leans strongly towards preserving the separation of logic and presentation, therefore ensures a perfect MVC - Model-View-Controller (page 86) - design, and ready to consume SOA services.

Mustache is intentionally constrained in the features it supports and, as a result, applications tend to require quite a bit of code to instantiate a template: all the application logic will be defined within the Controller code, not in the View source. This may not be to everybody's tastes. However, while this design limits the power of the template language, it does not limit the power or flexibility of the template system. This system supports arbitrarily complex text formatting.

Finally, Mustache is designed with an eye towards efficiency. Template instantiation is very quick, with an eye towards minimizing both memory use and memory fragmentation. As a result, it sounds like a perfect template system for our mORMot framework.

18.2.2.2. Mustache principles

There are two main parts to the Mustache template system:
- Templates (which are plain text files);
- Data dictionaries (aka Context).

For instance, given the following template:

```html
<h1>{{header}}</h1>

{{#items}}
{{#first}}
<li><strong>{{name}}</strong></li>
{{/first}}
{{#link}}
<li><a href="{{url}}">{{name}}</a></li>
{{/link}}
{{/items}}
```
and the following data context:

```
{
  "header": "Colors",
  "items": [
    {"name": "red", "first": true, "url": "#Red"},
    {"name": "green", "link": true, "url": "#Green"},
    {"name": "blue", "link": true, "url": "#Blue"}
  ],
  "empty": true
}
```

The Mustache engine will render this data as such:

```
<h1>Colors</h1>
<li><strong>red</strong></li>
<li><a href="#Green">green</a></li>
<li><a href="#Blue">blue</a></li>
<p>The list is empty.</p>
```

In fact, you did not see any "if" nor "for" loop in the template, but Mustache conventions make it easy to render the supplied data as the expected HTML output. It is up to the MVC Controller to render the data as expected by the template, e.g. for formatting dates or currency values.

### 18.2.2.3. Mustache templates

The Mustache template logic-less language has five types of tags:
- Variables;
- Sections;
- Inverted Sections;
- Comments;
- Partials.

All those tags will be identified with mustaches, i.e. `{...}`. Anything found in a template of this form is interpreted as a template marker. All other text is considered formatting text and is output verbatim at template expansion time.

<table>
<thead>
<tr>
<th>Marker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{{variable}}</td>
<td>The variable name will be searched recursively within the current context (possibly with dotted names), and, if found, will be written as escaped HTML. If there is no such key, nothing will be rendered.</td>
</tr>
<tr>
<td>{{{variable}}}</td>
<td>The variable name will be searched recursively within the current context, and, if found, will be written directly, without any HTML escape.</td>
</tr>
<tr>
<td>{{&amp; variable}}</td>
<td>If there is no such key, nothing will be rendered.</td>
</tr>
</tbody>
</table>
Defines a block of text, aka *section*, which will be rendered depending of the section variable value, as searched in the current context:
- If section equals false or is an *empty list* [], the whole block won't be rendered;
- If section is non-false but not a list, it will be used as the context for a single rendering of the block;
- If section is a non-empty list, the text in the block will be rendered once for each item in the list - the context of the block will be set to the current item for each iteration.

Defines a block of text, aka *inverted section*, which will be rendered depending of the section variable *inverted* value, as searched in the current context:
- If section equals false or is an *empty list*, the whole block will be rendered;
- If section is non-false or a non-empty list, it won't be rendered.

The comment text will just be ignored.

The partial name will be searched within the registered *partials list*, then will be executed at run-time (so recursive partials are possible), with the current execution context.

The delimiters (i.e. by default `{{ . . }}`) will be replaced by the specified characters (may be convenient when double-braces may appear in the text).

In addition to those standard markers, the mORMot implementation of Mustache features:

<table>
<thead>
<tr>
<th>Marker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{{helperName value}}</code></td>
<td><em>Expression Helper</em>, able to change the value on the fly, before rendering. It could be used e.g. to display dates as text from TDateTime or TTimeLog values.</td>
</tr>
<tr>
<td><code>{{.}}</code></td>
<td>This pseudo-variable refers to the context object itself instead of one of its members. This is particularly useful when iterating over lists.</td>
</tr>
<tr>
<td><code>{{-index}}</code></td>
<td>This pseudo-variable returns the current item number when iterating over lists, starting counting at 1 (<code>{{-index0}}</code> will start counting at 0)</td>
</tr>
<tr>
<td><code>{{#-first}}</code></td>
<td>Defines a block of text (pseudo-section), which will be rendered - or <em>not</em> rendered for inverted <code>{{^-first}}</code> - for the <em>first</em> item when iterating over lists</td>
</tr>
<tr>
<td><code>{{/first}}</code></td>
<td></td>
</tr>
<tr>
<td><code>{{#-last}}</code></td>
<td>Defines a block of text (pseudo-section), which will be rendered - or <em>not</em> rendered for inverted <code>{{^-last}}</code> - for the <em>last</em> item when iterating over lists</td>
</tr>
<tr>
<td><code>{{/last}}</code></td>
<td></td>
</tr>
<tr>
<td><code>{{#-odd}}</code></td>
<td>Defines a block of text (pseudo-section), which will be rendered - or <em>not</em> rendered for inverted <code>{{^-odd}}</code> - for the <em>odd</em> item number when iterating over lists: it can be very useful e.g. to display a list with alternating row colors</td>
</tr>
<tr>
<td><code>{{/odd}}</code></td>
<td></td>
</tr>
<tr>
<td><code>{{&lt;partial}}</code></td>
<td>Defines an in-lined <em>partial</em> - to be called later via <code>{{&gt;partial}}</code> - within the scope of the current template</td>
</tr>
<tr>
<td><code>{{/partial}}</code></td>
<td></td>
</tr>
<tr>
<td><code>{{&quot;some text}}</code></td>
<td>This pseudo-variable will supply the given text to a callback, which will for instance transform its content (e.g. translate it), before writing it to the output</td>
</tr>
</tbody>
</table>
This set of markers will allow to easily write any kind of content, without any explicit logic nor nested code. As a major benefit, the template content could be edited and verified without the need of any Mustache compiler, since all those {{...}} markers will identify very clearly the resulting layout.

18.2.2.3.1. Variables
A typical Mustache template:

```
Hello {{name}}
You have just won {{value}} dollars!
Well, {{taxed_value}} dollars, after taxes.
```

Given the following hash:

```
{
  "name": "Chris",
  "value": 10000,
  "taxed_value": 6000
}
```

Will produce the following:

```
Hello Chris
You have just won 10000 dollars!
Well, 6000 dollars, after taxes.
```

You can note that {{variable}} tags are escaped for HTML by default. This is a mandatory security feature. In fact, all web applications which create HTML documents can be vulnerable to Cross-Site-Scripting (XSS) attacks unless data inserted into a template is appropriately sanitized and/or escaped. With Mustache, this is done by default. Of course, you can override it and force to not-escape the value, using {{{variable}}} or {{& variable}}.

For instance:

<table>
<thead>
<tr>
<th>Template</th>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>* {{name}}</td>
<td>{ &quot;name&quot;: &quot;Chris&quot;, &quot;value&quot;: 10000, &quot;taxed_value&quot;: 6000 }</td>
<td>* Chris</td>
</tr>
<tr>
<td>* {{age}}</td>
<td>&quot;name&quot;: &quot;Chris&quot;, &quot;value&quot;: 10000, &quot;taxed_value&quot;: 6000</td>
<td>*</td>
</tr>
<tr>
<td>* {{company}}</td>
<td>&quot;company&quot;: &quot;&lt;b&gt;GitHub&lt;/b&gt;&quot;</td>
<td>* &lt;b&gt;GitHub&lt;/b&gt;</td>
</tr>
<tr>
<td>* {{company}}}</td>
<td>}</td>
<td>* &lt;b&gt;GitHub&lt;/b&gt;</td>
</tr>
</tbody>
</table>

Variables resolve names within the current context with an optional dotted syntax, for instance:

<table>
<thead>
<tr>
<th>Template</th>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>* {{people.name}}</td>
<td>{ &quot;people&quot;: { &quot;name&quot;: &quot;Chris&quot;, &quot;value&quot;: 10000, &quot;taxed_value&quot;: 6000 } }</td>
<td>* Chris</td>
</tr>
<tr>
<td>* {{people.age}}</td>
<td>&quot;name&quot;: &quot;Chris&quot;, &quot;value&quot;: 10000, &quot;taxed_value&quot;: 6000</td>
<td>*</td>
</tr>
<tr>
<td>* {{people.company}}</td>
<td>&quot;company&quot;: &quot;&lt;b&gt;GitHub&lt;/b&gt;&quot;</td>
<td>* &lt;b&gt;GitHub&lt;/b&gt;</td>
</tr>
<tr>
<td>* {{people.company}}</td>
<td>}</td>
<td>* &lt;b&gt;GitHub&lt;/b&gt;</td>
</tr>
</tbody>
</table>

18.2.2.3.2. Sections

Sections render blocks of text one or more times, depending on the value of the key in the current context.

In our "wining template" above, what happen if we do want to hide the tax details?
In most script languages, we may write an if ... block within the template. This is what Mustache avoids. So we define a section, which will be rendered on need.
The template becomes:

```
Hello {{name}}
You have just won {{value}} dollars!
{{#in_ca}}
Well, {{taxed_value}} dollars, after taxes.
{{/in_ca}}
```

Here, we created a new section, named `in_ca`.

Given the hash value of `in_ca` (and its presence), the section will be rendered, or not:

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
</table>
| ```
  "name": "Chris",
  "value": 10000,
  "taxed_value": 6000,
  "in_ca": true
``` | Hello Chris
You have just won 10000 dollars!
Well, 6000 dollars, after taxes. |
| ```
  "name": "Chris",
  "value": 10000,
  "taxed_value": 6000,
  "in_ca": false
``` | Hello Chris
You have just won 10000 dollars! |
| ```
  "name": "Chris",
  "value": 10000,
  "taxed_value": 6000
``` | Hello Chris
You have just won 10000 dollars! |

Sections also change the context of its inner block. It means that the section variable content becomes the top-most context which will be used to identify any supplied variable key.

Therefore, the following context will be perfectly valid: we can define `taxed_value` as a member of `in_ca`, and it will be rendered directly, since it is part of the new context:

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
</table>
| ```
  "name": "Chris",
  "value": 10000,
  "in_ca": {
    "taxed_value": 6000
  }
``` | Hello Chris
You have just won 10000 dollars!
Well, 6000 dollars, after taxes. |
| ```
  "name": "Chris",
  "value": 10000,
  "taxed_value": 6000
``` | Hello Chris
You have just won 10000 dollars! |
In the latest context above, there are two `taxed_value` variables. The engine will use the one defined by the context in the `in_ca` section, i.e. `in_ca.taxed_value`; the one defined at the root context level (which equals 3000) is just ignored.

If the variable pointed by the section name is a list, the text in the block will be rendered once for each item in the list. The context of the block will be set to the current item for each iteration.

In this way we can loop over collections. Mustache allows any depth of nested loops (e.g. any level of master/details information).

<table>
<thead>
<tr>
<th>Template</th>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
</table>
| {{#repo}}
  {{repo}}: [{{name}} = "resque",
            {{name}} = "hub",
            {{name}} = "rip"
  ]
| {{/repo}} |            | <b>resque</b> <b>hub</b> <b>rip</b> |
| {{#repo}}
  {{repo}}: ["resque", "hub", "rip"]
| {{/repo}} |            | <b>resque</b> <b>hub</b> <b>rip</b> |

The latest template makes use of the `{{.}}` pseudo-variable, which allows to render the current item of the list.

### 18.2.2.3.3. Inverted Sections

An inverted section begins with a caret (^) and ends as a standard (non-inverted) section. They may render text once, based on the inverse value of the key. That is, the text block will be rendered if the key doesn’t exist, is false, or is an empty list.

Inverted sections are usually defined after a standard section, to render some message in case no information will be written in the non-inverted section:

<table>
<thead>
<tr>
<th>Template</th>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
</table>
| {{#repo}}
  <b>{{.}}</b>
| {{/repo}} |            | No repos :( |

### 18.2.2.3.4. Partials
Partials are some kind of external sub-templates which can be included within a main template, for instance to follow the same rendering at several places. Just like functions in code, they do ease template maintainability and spare development time.

Partials are rendered at runtime (as opposed to compile time), so recursive partials are possible. Just avoid infinite loops. They also inherit the calling context, so can easily be re-used within a list section, or together with plain variables.

In practice, partials shall be supplied together with the data context - they could be seen as "template context".

For example, this "main" template uses a {{> user}} partial:

```html
<h2>Names</h2>
{{#names}}{{> user}}{{/names}}
```

With the following template registered as "user":

```html
<strong>{{name}}</strong>
```

Can be thought of as a single, expanded template:

```html
<h2>Names</h2>
{{#names}}{{> user}}{{/names}}
{{name}}</strong>
```

In mORMot's implementations, you can also create some internal partials, defined as {{<partial}} ... {{/partial}} pseudo-sections. It may decrease the need of maintaining multiple template files, and refine the rendering layout.

For instance, the previous template may be defined at once:

```html
<h2>Names</h2>
{{#names}}{{> user}}{{/names}}

{{<user}}{{name}}</strong>{{/user}}
```

The same file will define both the partial and the main template. Note that we defined the internal partial after the main template, but we may have defined it anywhere in the main template logic: internal partials definitions are ignored when rendering the main template, just like comments.

**18.2.2.4. SynMustache unit**

Part of our mORMot framework, we implemented an optimized Mustache template engine in the SynMustache unit:

- It is the first Delphi implementation of Mustache;
- It has a separate parser and renderer (so you can compile your templates ahead of time);
- The parser features a shared cache of compiled templates;
- It passes all official Mustache specification tests, as defined at http://github.com/mustache/spec_ - including all weird whitespace process;
- External partials can be supplied as TSynMustachePartials dictionaries;
- {{.}},{{-index}} and {{"some text"}} pseudo-variables were added to the standard Mustache syntax;
- {{#first}}, {{#last}} and {{#odd}} pseudo-sections were added to the standard *Mustache* syntax;
- Internal partials can be defined via {{<partial>}} - also a nice addition to the standard *Mustache* syntax;
- It allows the data context to be supplied as JSON or our *TDocVariant* custom variant type (page 112);
- Almost no memory allocation is performed during the rendering;
- It is natively UTF-8, from the ground up, with optimized conversion of any string data;
- Performance has been tuned and grounded in SynCommons.pas's optimized code;
- Each parsed template is thread-safe and re-entrant;
- It follows the *Open/Closed principle* - see SOLID design principles (page 389) - so that any aspect of the process can be customized and extended (e.g. for any kind of data context);
- It is perfectly integrated with the other bricks of our mORMot framework, ready to implement dynamic web sites with true *Model-View-Controller* (page 86) design, and full separation of concerns in the views written in *Mustache*, the controllers being e.g. interface-based services - see *Client-Server services via interfaces* (page 419), and the models being our *Object-Relational Mapping (ORM)* (page 92) classes;
- API is flexible and easy to use.

### 18.2.2.4.1. Variables

Now, let's see some code.

First, we define our needed variables:

```pascal
var mustache: TSynMustache;
doc: TDocVariant;
```

In order to parse a template, you just need to call:

```pascal
mustache := TSynMustache.Parse(
  'Hello {{name}}
  You have just won {{value}} dollars!');
```

It will return a compiled instance of the template.

The `Parse()` class method will use the shared cache, so you won't need to release the `mustache` instance once you are done with it: no need to write a `try ... finally mustache.Free; end block.`

You can use a `TDocVariant` to supply the context data (with late-binding):

```pascal
TDocVariant.New(doc);
doc.name := 'Chris';
doc.value := 10000;
```

As an alternative, you may have defined the context data as such:

```pascal
doc := _ObjFast(['name','Chris','value',1000]);
```

Now you can render the template with this context:

```pascal
html := mustache.Render(doc);
// now html='Hello Chris'
// 'You have just won 10000 dollars!'
```

If you want to supply the context data as JSON, then render it, you may write:

```pascal
mustache := TSynMustache.Parse(
  'Hello {{value.name}}
  You have just won {{value.value}} dollars!');
html := mustache.RenderJSON('{{value:{name:"Chris",value:10000}}'});
// now html='Hello Chris'
// 'You have just won 10000 dollars!'
```

Note that here, the JSON is supplied with *MongoDB*-like extended syntax (i.e. field names are unquoted), and that `TSynMustache` is able to identify a dotted-named variable within the execution context.
As an alternative, you could use the following syntax to create the data context as JSON, with a set of parameters, therefore easier to work with in real code storing data in variables (for instance, any string variable is quoted as expected by JSON, and converted into UTF-8):

```go
mustache := TSynMustache.Parse('Hello {{name}}')
html := mustache.RenderJSON('name', 'Chris', '10000');
html='Hello Chris'
```

You can find in the mORMot.pas unit the `ObjectToJSON()` function which is able to transform any `TPersistent` instance into valid JSON content, ready to be supplied to a `TSynMustache` compiled instance.

If the object’s published properties have some getter functions, they will be called on the fly to process the data (e.g. returning 'FirstName Name' as `FullName` by concatenating both sub-fields).

**18.2.2.4.2. Sections**

Sections are handled as expected:

```go
mustache := TSynMustache.Parse('Shown.{{#person}}As {{name}}!{{/person}}end')
html := mustache.RenderJSON('person', 'true')
// now html='Shown.end'
```

Note that the sections change the data context, so that within the `{{person}}` section, you can directly access to the data context `person` member, i.e. writing directly `{{name}}`.

It supports also inverted sections:

```go
mustache := TSynMustache.Parse('Shown.{{^person}}Never shown!{{/person}}end')
html := mustache.RenderJSON('person', 'true')
// now html='Shown.end'
```

To render a list of items, you can write for instance (using the `{{.}}` pseudo-variable):

```go
mustache := TSynMustache.Parse('{{#things}}{{.}}{{/things}}')
html := mustache.RenderJSON('things', ['one', 'two', 'three'])
// now html='onetwothree'
```

The `{{-index}}` pseudo-variable allows to numerate the list items, when rendering:

```go
mustache := TSynMustache.Parse('My favorite things:
  1. {{things}}
  2. {{things}}
  3. {{things}}')
html := mustache.RenderJSON('things', ['Peanut butter', 'Pen spinning', 'Handstands'])
// now html='My favorite things: 1. Peanut butter 2. Pen spinning 3. Handstands'
```

**18.2.2.4.3. Partials**

External partials (i.e. standard Mustache partials) can be defined using `TSynMustachePartials`. You can define and maintain a list of `TSynMustachePartials` instances, or you can use a one-time partial, for a given rendering process, as such:

```go
mustache := TSynMustache.Parse('{{>partial}}')
html := mustache.RenderJSON('{}', TSynMustachePartials.CreateOwned([partical, '1', '2']))
// now html='12','external partials'
```

Here `TSynMustachePartials.CreateOwned()` expects the partials to be supplied as name/value pairs.

Internal partials (one of the SynMustache extensions), can be defined directly in the main template:

```go
mustache := TSynMustache.Parse('{{<partial}}1{{name}}{{/partial}}{{>partial}}4')
html := mustache.RenderJSON('name', '3')
// now html='134','internal partials'
```
### 18.2.2.4.4. Expression Helpers

*Expression Helpers* are an extension to the standard *Mustache* definition. They allow to define your own set of functions which will be called during the rendering, to transform one value from the context into a value to be rendered.

`TSynMustache.HelpersGetStandardList` will return a list of standard static helpers, able to convert `TDateTime` or `TTimeLog` values into text, or convert any value into its JSON representation. The current list of registered helpers are `DateTimeToText`, `DateToText`, `DateFmt`, `TimeLogToText`, `BlobToBase64`, `JSONQuote`, `JSONQuoteURI`, `ToJSON`, `EnumTrim`, `EnumTrimRight`, `PowerOfTwo`, `Equals`, `If`, `MarkdownToHtml`, `SimpleToHtml` and `WikiToHtml`. For instance, `{{TimeLogToText CreatedAt}}` will convert a `TCreateTime` field value into ready-to-be-displayed text.

The mustache tag syntax is `{{helper name value}}`. The supplied `value` parameter may be a variable name in the current context, or could be a constant number (`{{helpername 123}}`), a constant JSON string (`{{helpername "constant text"}}`), a JSON array (`{{helpername [1,2,3]}}`) or a JSON object (`{{helpername {name:"john",age:24}}}`). The value could be also a comma-separated set of values, which will be translated into a corresponding JSON array, the values being extracted from the current context, as with `{{DateFmt DateValue,"dd/mm/yyyy"}}`.

You could call recursively the helpers, just like you nest functions: `{{helper1 helper2 value}}` will call helper2 with the supplied value, which result will be passed as value to helper1.

But you can create your own list of registered *Expression Helpers*, even including some business logic, to compute any data during rendering, via `TSynMustache.HelperAdd` methods.

The helper should be implemented with such a method:

```pascal
class procedure TSynMustache.JSONQuote(const Value: variant; out result: variant);
var json: RawUTF8;
begin
  QuotedStrJSON(VariantToUTF8(Value),json);
  RawUTF8ToVariant(json,result);
end;
```

Here, the supplied `Value` parameter will be either from a variable of the context, or a constant, from JSON number, string, array or object - encoded as a `TDocVariant custom variant type` (page 112).

If the parameters were supplied as a comma-separated list, you may write multi-parameter functions as such:

```pascal
class procedure TSynMustache.DateFmt(const Value: variant; out result: variant);
begin
  with _Safe(Value)^ do
  if (Kind=dvArray) and (Count=2) and (TVarData(Values[0]).VType=varDate) then
    result := FormatDateTime(Values[1],TVarData(Values[0]).VDate) else
    SetVariantNull(result);
end;
```

So you could use such expression helper this way:

```
La date courante en France est : {{DateFmt DateValue,"dd/mm/yyyy"}}
```

The `Equals` helper is defined as such:

```pascal
class procedure TSynMustache.Equals_(const Value: variant; out result: variant);
begin
  // {{#Equals .,12}}
  with _Safe(Value)^ do
  if (Kind=dvArray) and (Count=2) and (SortDynArrayVariant(Values[0],Values[1])=0) then
    result := true else
    SetVariantNull(result);
end;
```

So you could use such expression helper this way:

```
La date courante en France est : {{DateFmt DateValue,"dd/mm/yyyy"}}
```

The `Equals` helper is defined as such:

```pascal
class procedure TSynMustache.Equals_(const Value: variant; out result: variant);
begin
  // {{#Equals .,12}}
  with _Safe(Value)^ do
  if (Kind=dvArray) and (Count=2) and
    (SortDynArrayVariant(Values[0],Values[1])=0) then
    result := true else
    SetVariantNull(result);
end;
```
You may use it in your template to provide additional view logic:

```mustache
{{#Equals Category,"Admin"}}
Welcome, Mister Administrator!
{{/Equals}}
```

The section ending may optionally only contain the helper name, so the following syntax is also correct, and perhaps less error-prone:

```mustache
{{#Equals Category,"Admin"}}
Welcome, Mister Administrator!
{{/Equals}}
```

The #If helper is even more powerful, since it allows to define some view logic, via $= \le \ge \lt \gt \lt\gt$ operators set between two values:

```mustache
{{#if .,"=",6}} Welcome, number six! {{/if}}
{{#if Total,">",1000}} Thanks for your income: your loyalty will be rewarded. {{/if}}
{{#if info,"=",""}} Warning: {{info}} {{/if}}
```

As an alternative, you could just put the operator without a string parameter:

```mustache
{{#if .=6}} Welcome, number six! {{/if}}
{{#if Total>1000}} Thanks for your income: your loyalty will be rewarded. {{/if}}
{{#if info>""}} Warning: {{info}} {{/if}}
```

This latest syntax may it pretty convenient to work with. Of course, since Mustache is expected to be a logic-less templating engine, you should better not use the #if helper in most cases, but rather add some dedicated flags in the supplied data context:

```mustache
{{isNumber6}} Welcome, number six! {{/isNumber6}}
{{showLoyaltyMessage}} Thanks for your income: your loyalty will be rewarded. {{/showLoyaltyMessage}}
{{showWarning}} Warning: {{info}} {{/showWarning}}
```

Helpers can be used to convert some wiki or markdown content into plain HTML, for instance, in the MVC blog sample, a ContentHtml boolean flag defines if a content (here the abstract text field) is already HTML-encoded, or if it needs to be converted via the WikiToHtml helper:

```mustache
{{#ContentHtml}}{{{abstract}}}{{/ContentHtml}}{{^ContentHtml}}{{{WikiToHtml abstract}}}{{/ContentHtml}}
```

The framework also offers some built-in optional Helpers tied to its ORM, if you create a MVC web application using mORMotMVC.pas - see below (page 519) - you can register a set of Expression Helpers to let your Mustache view retrieve a given TSQLRecord, from its ID, or display a given instance fields in an auto-generated table.

For instance, you may write:

```mustache
aMVCMustacheView.RegisterExpressionHelpersForTables(aRestServer,[TSQLMyRecord]);
```

This will define two Expression Helpers for the specified table:
- Any `{{#TSQLMyRecord MyRecordID}} ... {{/TSQLMyRecord MyRecordID}}` Mustache tag will read a TSLQMyRecord from the supplied ID value and put its fields in the current rendering data context, ready to be displayed in the view.
- Any `{{TSQLMyRecord.HtmlTable MyRecord}}` Mustache tag which will create a HTML table containing all information about the supplied MyRecord fields (from the current data context), with complex field handling (like TDateTime, TTimeLog, sets or enumerations), and proper display of the field names (and i18n).

### 18.2.4.5. Internationalization

You can define `"some text"` pseudo-variables in your templates, which text will be supplied to a callback, ready to be transformed on the fly: it may be convenient for i18n of web applications.
By default, the text will be written directly to the output buffer, but you can define a callback which may be used e.g. for text translation:

```delphi
procedure TTestLowLevelTypes.MustacheTranslate(var English: string);
begin
  if English='Hello' then
    English := 'Bonjour'
  else if English='You have just won' then
    English := 'Vous venez de gagner';
end;
```

Of course, in a real application, you may assign one TLanguageFile.Translate(var English: string) method, as defined in the mORMot18n.pas unit.

Then, you will be able to define your template as such:

```delphi
mustache := TSynMustache.Parse('Hello {{name}}
    You have just won
    {{value}}
    dollars!';
html := mustache.RenderJSON('{name:?,value:?}',[],['Chris',10000],MustacheTranslate);
```

All text has indeed been translated as expected.

### 18.2.2.5. Low-level integration with method-based services

You can easily integrate the Mustache template engine with the framework's ORM. To avoid any unneeded temporary conversion, you can use the TSQLRest.RetrieveDocVariantArray() method, and provide its TDocVariant result as the data context of TSynMustache.Render().

For instance, you may write, in any method-based service - see [Client-Server services via methods](page 373):

```delphi
var
template: TSynMustache;
  html: RawUTF8;
...
  template := TSynMustache.Parse('<ul>{{#items}}<li>{{Name}} was born on {{BirthDate}}</li>{{/items}}</ul>');</n
  html := template.Render(aClient.RetrieveDocVariantArray(TSQLBaby,'items','Name,BirthDate'));
```

Of course, this TSQLRest.RetrieveDocVariantArray() method accepts an optional WHERE clause, to be used according to your needs. You may even use paging, to split the list in smaller pieces.

Following this low-level method-based services process, you can easily create a high performance web server using mORMot, following the MVC pattern as such:

<table>
<thead>
<tr>
<th>MVC</th>
<th>mORMot</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model</strong></td>
<td><strong>Object-Relational Mapping (ORM)</strong> (page 92) and its TSQLModel / TSQLRecord definitions</td>
</tr>
<tr>
<td><strong>View</strong></td>
<td><strong>Mustache template engine</strong> (page 505) (may be stored as separated files or within the database)</td>
</tr>
<tr>
<td><strong>Controller</strong></td>
<td><strong>Method-based services - see [Client-Server services via methods](page 373)</strong></td>
</tr>
</tbody>
</table>

But still, a lot of code is needed to glue the MVC parts.

### 18.2.2.6. MVC/MVVM Design

In practice, the method-based services MVC pattern is difficult to work with. You have a lot of
plumbing to code by yourself, e.g. parameter marshalling, rendering or routing.

The mORMotMVC.pas unit offers a true MVVM (Model View ViewModel) design, much more advanced, which relies on interface definitions to build the application - see Interfaces (page 385):

<table>
<thead>
<tr>
<th>MVVM</th>
<th>mORMot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Object-Relational Mapping (ORM) (<a href="#">page 92</a>) and its TSQLModel / TSQLRecord definitions</td>
</tr>
<tr>
<td>View</td>
<td>Mustache template engine (<a href="#">page 505</a>) (may be stored as separated files or within the database)</td>
</tr>
<tr>
<td>ViewModel</td>
<td>Interface services - see also Client-Server services via interfaces (<a href="#">page 419</a>)</td>
</tr>
</tbody>
</table>

In the MVVM pattern, both Model and View components do match the classic Model-View-Controller ([page 86](#)) layout. But the ViewModel will define some kind of "model for the view", i.e. the data context to be sent and retrieved from the view.

In the mORMot implementation, interface methods are used to define the execution context of any request, following the convention over configuration pattern of our framework. In fact, the following conventions are used to define the ViewModel:

<table>
<thead>
<tr>
<th>ViewModel</th>
<th>mORMot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Route</td>
<td>From the interface name and its method name</td>
</tr>
<tr>
<td>Command</td>
<td>Defined by the method name</td>
</tr>
<tr>
<td>Controller</td>
<td>Defined by the method implementation</td>
</tr>
<tr>
<td>ViewModel Context</td>
<td>Transmitted by representation, as JSON, including complex values like TSQLRecord, records, dynamic arrays or variants (including TDocVariant)</td>
</tr>
<tr>
<td>Input Context</td>
<td>Transmitted as method input parameters (const/var) from the View</td>
</tr>
<tr>
<td>Output Context</td>
<td>Method output parameters (var/out) are sent to the View</td>
</tr>
<tr>
<td>Actions</td>
<td>A method will render the associated view with the output parameters, or go to another command (optionally via EMVCApplication)</td>
</tr>
</tbody>
</table>

This may sound pretty unusual (if you are coming from a RubyOnRails, AngularJS, Meteor or .Net implementations), but it has been identified to be pretty convenient to use. Main benefit is that you do not need to define explicit data structures for the ViewModel layer. The method parameters will declare the execution context for you at interface level, ready to be implemented in a TMVCApplication class. In practice, this implementation uses the interface input and output parameters are an alternate way to define the $scope content of an AngularJS application.

The fact that the ViewModel data context is transmitted as JSON content - by representation just like REST - see REST ([page 311](#)) - allows nice side effects:
- Views do not know anything about the execution context, so are very likely to be uncoupled from any business logic - this will enhance security and maintainability of your applications;
- You can optionally see in real time the JSON data context (by using a fake root/methodname/json URI) of a running application, for easier debugging of the Controller or the Views;
- You can test any View by using fake static JSON content, without the need of a real server;
- In fact, Views could be even not tied to the web model, but run in a classic rich application, with VCL/FMX User Interface (we still need to automate the binding process to UI components, but this is technically feasible, whereas almost no other MVC web framework do support this);
- Since Interface are used to define the Controller, you could mock and stub them - see Interfaces in practice: dependency injection, stubs and mocks (page 406) - for proper unit testing;
- In the Controller code, you have access to the mORMot ORM methods and services to write the various commands, making it pretty easy to implement a web front-end to any SOA project (also sharing a lot of high-level Domain types);
- The associated data Model is mORMot's ORM, which is also optimized for JSON processing, so most of memory fragmentation is reduced to the minimum during the rendering (see e.g. the use of RawJSON below);
- The Controller will be most of the time hosted within the web server application, but may be physically hosted in another remote process - this remote Controller service may even be shared between web and VCL/FMX clients;
- Several levels of cache could be implemented, based on the JSON content, to leverage the server resources and scale over a huge number of clients.

The next chapter will uncover how to build such solid MVC / MVVM Web Applications using mORMot.
19. MVC/MVVM Web applications

We will now explain how to build a MVC/MVVM web application using mORMot, starting from the "30 - MVC Server" sample. Following explanations may be a bit unsynchronized from the current state of the sample source code in the "unstable" branch of the framework repository, but you will get here below the main intangible points.

This little web application publishes a simple BLOG, not fully finished yet (this is a Sample, remember!). But you can still execute it in your desktop browser, or any mobile device (thanks to a simple Bootstrap-based responsive design), and see the articles list, view one article and its comments, view the author information, log in and out.

This sample is implemented as such:

<table>
<thead>
<tr>
<th>MVVM</th>
<th>Source</th>
<th>mORMot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>MVCModel.pas</td>
<td>TSQLRestServerDB ORM over a SQLite3 database</td>
</tr>
<tr>
<td>View</td>
<td>*.html</td>
<td>Mustache template engine (page 505) in the Views sub-folder</td>
</tr>
<tr>
<td>ViewModel</td>
<td>MVCViewModel.pas</td>
<td>Defined as one IBlogApplication interface</td>
</tr>
</tbody>
</table>

For the sake of the simplicity, the sample will create some fake data in its own local SQLite3 database, the first time it is executed.

19.1. MVCModel

19.1.1. Data Model

The MVCModel.pas unit defines the database Model, as regular TSQLRecord classes. For instance, you will find the following type definitions:

```pascal
TSQLContent = class(TSQLRecordTimestamped)
private ...
```
published
  property Title: RawUTF8 index 80 read fTitle write fTitle;
  property Content: RawUTF8 read fContent write fContent;
  property Author: TSQLAuthor read fAuthor write fAuthor;
  property AuthorName: RawUTF8 index 50 read fAuthorName write fAuthorName;
end;

TSQLArticle = class(TSQLContent)
private ...
public
  class function CurrentPublishedMonth: Integer;
  class procedure InitializeTable(Server: TSQLRestServer; const FieldName: RawUTF8;
     Options: TSQLInitializeTableOptions); override;
  procedure TagsAddOrdered(aTagID: integer; var aTags: TSQLTags);
  property PublishedMonth: Integer read fPublishedMonth write fPublishedMonth;
  property Abstract: RawUTF8 index 1024 read fAbstract write fAbstract;
  property Tags: TIntegerDynArray index 1 read fTags write fTags;
end;

TSQLComment = class(TSQLContent)
private ...
published
  property Article: TSQLArticle read fArticle write fArticle;
end;

Then the whole database model will be created in this function:

function CreateModel: TSQLModel;
begin
  result := TSQLModel.Create(['TSQLBlogInfo', 'TSQLAuthor', 'TSQLTag', 'TSQLArticle', 'TSQLComment', 'TSQLArticleSearch'], 'blog');
  TSQLArticle.AddFilterNotVoidText(['Title', 'Content']);
  TSQLComment.AddFilterNotVoidText(['Title', 'Content']);
  TSQLTag.AddFilterNotVoidText(['Ident']);
  result.Props[TSQLArticleSearch].FTS4WithoutContent(TSQLArticle);
end;

As you can discover:
- We used class inheritance to gather properties for similar tables;
- Some classes are not part of the model, since they are just abstract parents, e.g. TSQLContent is not part of the model, but TSQLArticle and TSQLComment are;
- We defined some regular one-to-one relationships, e.g. every Content (which may be either an Article or a Comment) will be tied to one Author - see "One to one" or "One to many" (page 151);
- We defined some regular one-to-many relationships, e.g. every Comment will be tied to one Article;
- Article tags are stored as a dynamic array of integer within the record, and not in a separated pivot table: it will make the database smaller, and queries faster (since we avoid a JOIN);
- Some properties are defined (and stored) twice, e.g. TSQLContent defines one AuthorName field in addition to the Author ID field, as a convenient direct access to the author name, therefore avoiding a JOINed query at each Article or a Comment display - see Shared nothing architecture (or sharding) (page 155);
- We defined the maximum expected width for text fields (e.g. via Title: RawUTF8 index 80), even if it won't be used by SQLite3 - it will ease any eventual migration to an external database, in the future - see Code-first ORM (page 268);
- Some validation rules are set using TSQLArticle.AddFilterNotVoidText() method, which will be applied before an article is stored in the controller's code (in TBlogApplication. ArticleCommit);
- The whole application will run without writing any SQL, but just high-level ORM methods;
- Even if we want to avoid writing SQL, we tried to modelize the data to fit regular RDBMS expectations, e.g. for most used queries (like the one run from the main page of the BLOG);
- Full Text indexation data, implemented as FTS3/FTS4/FTS5 (page 215) in the SQLite3 engine, is stored in a dedicated TSQLArticleSearch table - see FTS4 index tables without content (page 218) for details about this powerful feature.

Foreign keys and indexes are managed as such:
- The TSQLRecord.ID primary key of any ORM class will be indexed;
- For both one-to-one and one-to-many relationships, indexes are created by the ORM: for instance, TSQLArticle.Author and TSQLComment.Author will be indexed, just as TSQLComment.Article;
- A SQL index will be needed for TSQLArticle.PublishedMonth field, which is used to display a list of publication months in the main BLOG page, and link to the corresponding articles.

The following code will take care of it:

```pascal
class procedure TSQLArticle.InitializeTable(Server: TSQLRestServer; const FieldName: RawUTF8; Options: TSQLInitializeTableOptions);
begin
  inherited;
  if (FieldName='') or (FieldName='PublishedMonth') then
    Server.CreateSQLIndex(TSQLArticle,'PublishedMonth',false);
end;
```

19.1.2. Hosted in a REST server over HTTP

The ORM is defined to run over a SQLite3 database in the main MVCServer.dpr program, then served via a HTTP server as defined in MVCServer.dpr:

```pascal
aModel := CreateModel;
try
  aServer := TSQLRestServerDB.Create(aModel,ChangeFileExt(paramstr(0),'.db'));
  try
    aServer.DB.Synchronous := smNormal;
    aServer.Database.LockingMode := lmExclusive;
    aServer.CreateMissingTables;
    aApplication := TBlogApplication.Create;
    try
      aApplication.Start(aServer);
      aHTTPServer := TSQLHttpServer.Create('8092',aServer,'+',useHttpApiRegisteringURI);
      try
        aHTTPServer.RootRedirectToURI('blog/default'); // redirect server:8092
        writeln('"MVC Blog Server" launched on port 8092');
      ....
```
expected by the ORM/ SOA methods of the TSQLRestServer instance. For proper AJAX / JavaScript process, you may have to write:

```plaintext
aHttpServer.AccessControlAllowOrigin := '*'; // allow cross-site AJAX queries
```

Any attempt to access to the project.com or www.project.com URI will be redirected to the following method-based service:

```plaintext
procedure TMyServer.Html(Ctxt: TSQLRestServerURIContext);
begin
  if fMyFileCache='' then
    fMyFileCache := StringFromFile(ChangeFileExt(paramstr(0),'.html'));
  Ctxt.Returns(fMyFileCache,HTTP_SUCCESS,HTML_CONTENT_TYPE_HEADER,true);
end;
```

This method will serve some static HTML content as the main front end page of this server connected to the Internet. For best performance, this UTF-8 content is cached in memory, and the HTTP 304 command will be handled, if the browser supports it. Of course, your application may return some more complex content, even serving a set of files hosted in a local folder, e.g. by calling Ctxt.ReturnFile() or Ctxt.ReturnFileFromFolder() methods in this Html() service:

```plaintext
procedure TMyServer.Html(Ctxt: TSQLRestServerURIContext);
begin
  Ctxt.ReturnFileFromFolder('c:\www');
end;
```

This single method will search for any matching file in the local c:\www folder and its sub-directories, returning the default index.html content if no file is specified at URI level. See the optional parameters to the Ctxt.ReturnFileFromFolder() method for proper tuning, e.g. to change the default file name or disable the HTTP 304 answers. In all cases, the file content will be served by the High-performance http.sys server (page 326) directly from the kernel mode, so will be very fast.

In order to have the BLOG content hosted in root/blog URI, you should specify the expected sub-URI when initializing your TMVCApplication:

```plaintext
procedure TBlogApplication.Start(aServer: TSQLRestServer);
begin
  ...
  fMainRunner := TMVCRunOnRestServer.Create(self,nil,'blog').
  ...
```

Here, any request to blog.project.com will be redirected to root/blog, so will match the expected TBlogApplication URIs. Note that by default, TMVCRunOnRestServer.RunOnRestServerSub will redirect any root/blog request to root/blog/default, so this URI will be transparent for the user.

## 19.2. MVCViewModel

### 19.2.1. Defining the commands

The MVCViewModel.pas unit defines the Controller (or ViewModel) of the "30 - MVC Server" sample application. It uses the mORMotMVC.pas unit, which is the main MVC kernel for the framework, allowing to easily create Controllers binding the ORM/ SOA features (mORMot.pas) to the Mustache Views (SynMustache.pas).

First of all, we defined an interface, with the expected methods corresponding to the various commands of the web application:

```plaintext
IBlogApplication = interface(IMVCApplication)
  procedure ArticleView(
```
In fact, IMVCApplication is defined as such in mORMotMVC.pas:

```pascal
IMVCApplication = interface(IInvokable)
['{C48718BF-861B-448A-B593-8012DB51E15D}']
procedure Default(var Scope: variant);
procedure Error(var Msg: RawUTF8; var Scope: variant);
end;
```

As such, the IBlogApplication will define the following web pages, corresponding to each of its methods: Default, Error, ArticleView, AuthorView, Login, Logout, ArticleEdit and ArticleCommit. Each command of this application will map an URI, e.g. /blog/default or /blog/login - remember that our model defined 'blog' as its root URI. You may let all commands be accessible from a sub-URI (e.g. /blog/web/default), but here this is not needed, since we are creating a "pure web" application.

Each command will have its own View. For instance, you will find Default.html, Error.html or ArticleView.html in the "Views" sub-folder of the sample. If you did not supply any file in this folder, some void files will be created.

Incoming method parameters of each method (i.e. defined as const or var) will be transmitted on the URI, encoded as regular HTTP parameters, whereas outgoing method parameters (i.e. defined as var or out) will be transmitted to the View, as data context for the rendering. Simple types are transmitted (like integer or RawUTF8); but you will also find ORM classes (like TSQLAuthor), an outgoing TObjectList, or some variant - which may be either values or a complex TDocVariant custom variant type (page 112).

In fact, you may find out that the Login, Logout and ArticleCommit methods do not have any outgoing parameters, but were defined as function returning a TMVCAction record. This type is declared as such in mORMotMVC.pas:

```pascal
TMVCAction = record
    RedirectToMethodName: RawUTF8;
    RedirectToMethodParameters: RawUTF8;
    ReturnedStatus: cardinal;
end;
```

Any method returning a TMVCAction content won't render directly any view, but will allow to go directly to another method, for proper rendering, just by providing a method name and some optional parameters.

Note that even the regular views, i.e. the methods which do not have this TMVCAction parameter, may break the default rendering process on any error, raising an EMVCApplication exception which will in fact redirect the view to another page, mainly the Error page.

To better understand how it works, run the "30 - MVC Server" sample. Remember that to be able to register the port #8092 for the http.sys server, you will need to run the MVCServer.exe program at least once with Windows Administrator rights - see URI authorization as Administrator (page 327).
Then point your browser to http://localhost:8092/.. - you will see the main page of the BLOG, filled with some random data. Quite some "blabla", to be sincere!

What you see is the Default page rendered. The IBlogApplication.Default() method has been called, then the outgoing Scope data has been rendered by the Default.html Mustache template.

If you click on an article title, it will go to http://localhost:8092/blog/articleView?id=99.. - i.e. calling IBlogApplication.ArticleView() with the ID parameter containing 99, and other incoming parameters (i.e. WithComments and Direction) set to their default value (i.e. respectively false and 0). The ArticleView() method will then read the TSQLArticle data from the ORM, then send it to the ArticleView.html Mustache template.

Now, just change in your browser the URI from http://localhost:8092/blog/articleView?id=99.. (here we clicked on the Article with ID=99) into http://localhost:8092/blog/articleView/json?id=99.. (i.e. entering /articleView/json instead of /articleView, as a fake sub-URI).

Now the browser is showing you the JSON data context, as transmitted to the ArticleView.html template. Just check both the JSON content and the corresponding Mustache template: I think you will find out how it works. Take a look at Mustache template engine (page 505) as reference.

From any blog article view, click on the "Show Comments" button: you are redirected to a new page, at URI http://localhost:8092/blog/ArticleView?id=99&withComments=true#comments.. and now the comments corresponding to the article are displayed. If you click on the "Previous" or "Next" buttons, a new URI http://localhost:8092/blog/ArticleView?id=99&withComments=true&direction=1.. will be submitted: in fact, direction=1 will search for the previous article, and we still have the withComments=true parameter set, so that the user will be able to see the comments, as expected. If you click on the "Hide Comments" button, the URI will change to be without any withComments=true parameter - i.e. http://localhost:8092/blog/ArticleView?id=98#comments..: now the comments won't be displayed.

The sequence is rendered as such:
In this diagram, we can see that each HTTP request is stateless, uncoupled from the previous. The user experience is created by changing the URI with additional parameters (like `withComments=true`). This is how the web works.

Then try to go to `http://localhost:8092/blog/mvc-info..` and check out the page which appears. You will get all the information corresponding to your application, especially a list of all available commands:

```
/blog/Default?Scope=..[variant]..
/blog/Error?Msg=..[string]..&Scope=..[variant]..
/blog/ArticleView?ID=..[integer]..&withComments=..[boolean]..&Direction=..[integer]..
/blog/AuthorView?ID=..[integer]..
/blog/Login?LogonName=..[string]..&PlainPassword=..[string]..
/blog/Logout
/blog/ArticleEdit?ID=..[integer]..&Title=..[string]..&Content=..[string]..&ValidationError=..[variant]..
/blog/ArticleCommit?ID=..[integer]..&Title=..[string]..&Content=..[string]..
```

And every view, including its data context, e.g.

```
{{Main}}: variant
{{ID}}: integer
{{Author}}: TSQLAuthor
{{Articles}}: variant
```
You may use this page as reference when writing your Mustache Views. It will reflect the exact state of the running application.

19.2.2. Implementing the Controller

To build the application Controller, we will need to implement our IBlogApplication interface.

```pascal
TBlogApplication = class(TMVCApplication,IBlogApplication)
... public
  procedure Start(aServer: TSQLRestServer); reintroduce;
  procedure Default(var Scope: variant);
    out Article: TSQLArticle; out Author: variant;
    out Comments: TObjectList);
... end;
```

We defined a new class, inheriting from TMVCApplication - as defined in mORMotMVC.pas, and implementing our expected interface. TMVCApplication will do all the low-level plumbing for you, using a set of implementation classes.

Let's implement a simple command:

```pascal
procedure TBlogApplication.AuthorView(var ID: integer; out Author: TSQLAuthor;
  out Articles: variant);
begin
  RestModel.Retrieve(ID,Author);
  if Author.ID<>0 then
    Articles := RestModel.RetrieveListJSON(
      TSQLArticle,'Author=? order by id desc limit 50',[ID],ARTICLE_FIELDS) else
    raise EMVCApplication.CreateGotoError(HTTP_NOTFOUND);
end;
```

By convention, all parameters are allocated when TMVCApplication will execute a method. So you do not need to allocate or handle the Author: TSQLAuthor instance lifetime.

You have direct access to the underlying TSQLRest instance via TMVCApplication.RestModel: so all CRUD operations are available. You can let the ORM do the low level SQL work for you: to retrieve all information about one TSQLAuthor and get the list of its associated articles, we just use a TSQLRest method with the appropriate WHERE clause. Here we returned the list of articles as a TDocVariant, so that they will be transmitted as a JSON array, without any intermediate marshalling to TSQLArticle instances, but with the Tags dynamic array published property returned as an array of integers (you may have used TObjectList or RawJSON instead, as will be detailed below).

In case of any error, an EMVCApplication will be raised: when such an exception happens, the TMVCApplication will handle and convert it into a page change, and a redirection to the IBlogApplication.Error() method, which will return an error page, using the Error.html view template.

Let's take a look at a bit more complex method, which we talked about in mORMot MVC/MVVM URI - Commands sequence (page 525):

```pascal
procedure TBlogApplication.ArticleView(
  ID: integer; var WithComments: boolean; Direction: integer;
  out Article: TSQLArticle; out Author: variant; out Comments: TObjectList);
var newID: TID;
const WHERE: array[1..2] of PUTF8Char = ('ID<? order by id desc','ID>? order by id');
beginn
  if Direction in [1,2] then // allows fast paging using index on ID
    if RestModel.OneFieldValue(TSQLArticle, 'ID',WHERE[Direction],[],[ID],newID) and
```
(newID<>0) then
  ID := newID;
RestModel.Retrieve(ID,Article);
if Article.ID<>0 then begin
  Author := RestModel.RetrieveDocVariant(
    TSQLAuthor,'ID=?',[Article.Author.ID], 'FirstName,FamilyName');
  if WithComments then begin
    Comments.Free; // we will override the TObjectList created at input
    Comments := RestModel.RetrieveList(TSQLComment,'Article=?',[Article.ID]);
    end;
  end else
    raise EMVCApplication.CreateGotoError(HTTP_NOTFOUND);
end;

This method has to manage several use cases:
- Display an Article from the database;
- Retrieve the Author first name and family name;
- Optionally display the associated Comments;
- Optionally get the previous or next Article;
- Trigger an error in case of an invalid request.

Reading the above code is enough to understand how those 5 features are implemented in this method. The incoming parameters, as triggered by the Views, are used to identify the action to be taken. Then TMVCApplication.RestModel methods are used to retrieve the needed information directly from the ORM. Outgoing parameters (Article,Author,Comments) are transmitted to the Mustache View, for rendering.

In fact, there are several ways to retrieve your data, using the RestModel ORM methods. For instance, in the above code, we used a TObjectList to transmit our comments. But we may have used a TDocVariant custom variant type (page 112) parameter:

```pascal
procedure TBlogApplication.ArticleView(
  ID: integer; var WithComments: boolean; Direction: integer;
  out Article: TSQLArticle; out Author: variant; out Comments: variant);
...
if WithComments then
  Comments := RestModel.RetrieveDocVariantArray(TSQLComment,'','Article=?',[Article.ID],'');
```

In this case, data will be returned per representation, as variant values. Any dynamic array properties will be identified in the TSQLRecord, and converted as proper array of values.

Or with a RawJSON kind of output parameter:

```pascal
procedure TBlogApplication.ArticleView(
  ID: integer; var WithComments: boolean; Direction: integer;
  out Article: TSQLArticle; out Author: variant; out Comments: RawJSON);
...
if WithComments then
  Comments := RestModel.RetrieveListJSON(TSQLComment,'Article=?',[Article.ID],'');
```

Using a RawJSON will be in fact the fastest way of processing the information on the server side. But it will return the data directly from the database - as a consequence, dynamic arrays will be returned as a Base64-encoded blob.

It is up to you to choose the method and encoding needed for your exact context.

If your purpose is just to retrieve some data and push it back to the view, RawJSON is fast, but a TDocVariant will also convert dynamic arrays to a proper JSON array. If you want to process the returned information with some business code, returning a TObjectList may be convenient if you need to run some TSQLRecord methods on the returned list.

or a TDocVariant array may have its needs, if you want to create some meta-object gathering all
information, e.g. for Scope as returned by the Default method:

```delphi
procedure TBlogApplication.Default(var Scope: variant);
...
  if not fDefaultData.AddExistingProp('Archives',Scope) then
    fDefaultData.AddNewProp('Archives', RestModel.RetrieveDocVariantArray(
      TSQLArticle.'', 'group by PublishedMonth order by PublishedMonth desc limit 12', [],
      'distinct(PublishedMonth),max(ID)+1 as FirstID'), Scope);
end;
```

You can notice how the calendar months are retrieved from the database, using a safe fDefaultData: ILockedDocVariant private field to store the value as cache, in a thread-safe manner (we will see later more about how to implement thread-safety). If the 'Archives' value is in the fDefaultData cache, it will be returned immediately as part of the Scope returned document. Otherwise, it will use RestModel.RetrieveDocVariantArray to retrieve the last 12 available months. When a new Article is created, or modified, TBlogApplication.FlushAnyCache will call fDefaultData.Clear to ensure that the updated information will be retrieved from the database on next Default() call.

The above ORM request will generate the following SQL statement:

```sql
SELECT distinct(PublishedMonth),max(ID)+1 as FirstID FROM Article
  group by PublishedMonth order by PublishedMonth desc limit 12
```

The Default() method will therefore return the following JSON context:

```json
{
  "Scope": {
    ...
    "Archives": [
      { "PublishedMonth": 24178, "FirstID": 101 }
      { "PublishedMonth": 24177, "FirstID": 100 }
    ...
    ...
  }
```

... which will be processed by the Mustache engine.

If you put a breakpoint at the end of this Default() method, and inspect the "Scope" variable, the Delphi debugger will in fact show you in real time the exact JSON content, retrieved from the ORM.

I suspect you just find out how mORMot's ORM/ SOA abilites, and JSON / TDocVariant offer amazing means of processing your data. You have the best of both worlds: ORM/ SOA gives you fixed structures and strong typing (like in C++/CH/Ja), whereas TDocVariant gives you a flexible object scheme, using late-binding to access its content (like in Python/Ruby/JavaScript).

19.2.3. Variable input parameters

If you want to support a variable number of named parameters, you can define a variant input parameter, and provide the input as a JSON document, using a TDocVariant storage. But marshalling the context as JSON will involve using some JavaScript in the HTML page, which may not be very convenient.

If you want to handle a non-fixed set of regular URI or POST value, you can prefix all the incoming parameter names with the dotted name of a single defined variant. For instance, if you have the following controller method:

```delphi

```
Then you can supply as parameter at URI level:

\[
p.a1=5&p.a2=dfasdfa\]

And you will be able to handle them in the controller body:

```pascal
function TnWebMVCMenu.CadastroSalvar3(const p: variant): TMVCAction;
begin
  GotoView(result, 'Cadastro', ['pp1', p.a1, 'pp2', p.a2])
end;
```

You are now free to specify some versatile HTML forms in your views, and provide the controller with any kind of input parameters.
Of course, it may sound safer and easier to explicitly define and name each one of the input parameters, with simple types like `integer` or `RawUTF8`. But this convention may help you work with any kind of HTML views.

### 19.2.4. Using Services in the Controller

Any controller method could retrieve and execute any dependency from its interface, following the *IoC* pattern - see *Dependency Inversion Principle* (page 400).
You have two ways of performing the dependency resolution:
- From the associated TSQLRest.Services container;
- From its own protected `Resolve()` method, since `TMVCApplication` inherits from `TInjectableObject`.
In fact, you can set up your `TMVCApplication` instance to use any external dependencies, including stubs and mocks, or high-level DDD services (e.g. repository or modelization process), using its `CreateInjected()` constructor instead of plain `Create`.

### 19.2.5. Controller Thread Safety

When run from a TSQLRestServer instance, our MVC application commands will be executed by default without any thread protection. When hosted within a TSQLHttpServer instance - see *High-performance http.sys server* (page 326) - several threads may execute the same `Controller` methods at the same time. It is therefore up to your application code to ensure that your `TMVCApplication` process is thread safe.

Note that by design, all `TMVCApplication.RestModel ORM` methods are thread-safe.
If your `Controller` business code only uses ORM methods, sending back the information to the `Views`, without storing any data locally, it will be perfectly thread safe.
See for instance the `TBlogApplication.AuthorView` method we described above.
But consider this method (simplified from the real "30 - MVC Server" sample):

```pascal
type
  TBlogApplication = class(TMVCApplication, IBlogApplication)
  protected
    fDefaultArticles: variant;
    ...

procedure TBlogApplication.Default(var Scope: variant);
begin
  if VarIsEmpty(fDefaultArticles) then
    fDefaultArticles := RestModel.RetrieveDocVariantArray(
```
In fact, even if this method may sound safe, we have an issue when it is executed by several threads: one thread may be assigning a value to fDefaultArticles, whereas another thread may be using the fDefaultArticles content. This may result into random runtime errors, very difficult to solve. Even creating a local variable may not be safe, since any access to fDefaultArticles should be protected.

A first - and brutal - solution could be to force the TSQLRestServer instance to execute all method-based services (including our MVC commands) in a giant lock, as stated about Thread-safety (page 335):

```pascal
aServer.AcquireExecutionMode[execSOAByNameMethod] := amLocked; // or amBackgroundThread
```

But this may slow down the whole server process, and reduce its scaling abilities.

You could also lock explicitly the Controller method, for instance:

```pascal
procedure TBlogApplication.Default(var Scope: variant);
begin
  Locker.ProtectMethod;
  if VarIsEmpty(fDefaultData) then
    ...
end;
```

Using the TMVCApplication.Locker: IAutoLocker is a simple and efficient way of protecting your method. In fact, TAutoLocker class' ProtectMethod will return an IUnknown variable, which will let the compiler create an hidden try .. finally block in the method body, to release the lock when it quits. But this locker will be shared by the whole TMVCApplication instance, so will be like a giant lock on your Controller process.

A more tuned and safe implementation may be to use a ILockedDocVariant instead of a plain TDocVariant for caching the data. You may therefore write:

```pascal
type
  TBlogApplication = class(TMVCApplication,IBlogApplication)
  protected
    fDefaultData: ILockedDocVariant;
  ...

procedure TBlogApplication.Start(aServer: TSQLRestServer);
begin
  fDefaultData := TLockedDocVariant.Create;
  ...

procedure TBlogApplication.Default(var Scope: variant);
begin
  if not fDefaultData.AddExistingProp('Articles',Scope) then
    fDefaultData.AddNewProp('Articles',RestModel.RetrieveDocVariantArray(TSQLArticle,'','order by ID desc limit 20',[],ARTICLE_FIELDS),Scope);
end;
```

Using ILockedDocVariant will ensure that only access to this resource will be locked (no giant lock any more), and that slow ORM process (like RestModel.RetrieveDocVariantArray) will take place lock-free, to maximize the resource usage.

This is in fact the pattern used by the "30 - MVC Server" sample. Even Client-Server services via interfaces (page 419) may benefit from this TLockedDocVariant kind of storage, for efficient multi-thread process - see Server-side execution options (threading) (page 461).

### 19.2.6. Web Sessions

Sessions are usually implemented via cookies, in web sites. A login/logout procedure enhances security of the web application, and User experience can be tuned via small persistence of
client-driven data. The TMVCApplication class allows creating such sessions.

You can store whatever information you need within the client-side cookie. TMVCSessionWithCookies allows to define a record, which will be used to store the information as optimized binary, in the browser cache. You can use this cookie information as a cache to the current session, e.g. storing the logged user display name, his/her preferences or rights - avoiding a round trip to the database.

Of course, you should never trust the cookie content (even if our format uses secure encryption, and a digital signature via a HMAC-CRC32C algorithm). But you can use it as a convenient cache, always checking the real data in the database when you are about to perform any security-related action. The cookie also stores an integer Session ID, and issuing and expiration dates: as such, it matches all JWT (Javascript Web Token) - see http://jwt.io - features, as signature, encryption, and jwt/iat/exp claims, with a smaller overhead, and without using unsafe Web Local Storage.

For our "30 - MVC Server" sample application, we defined the following record in MVCViewModel.pas:

```delphi
TCookieData = packed record
  AuthorName: RawUTF8;
  AuthorID: cardinal;
  AuthorRights: TSQLAuthorRights;
end;
```

This record will be serialized in two ways:

- As raw binary, without the field names, within the cookie, after Base64 encoding of encrypted and digitally signed data;
- As a JSON object, with explicit field names, when transmitted to the Views as "Session" data context.

In order to have proper JSON serialization of the record, you will need to specify its structure, if you use a version of Delphi without the new RTII (i.e. before Delphi 2010) - see Record serialization (page 297).

Then we can use the TMVCApplication.CurrentSession property to perform the authentication, after successful login:

```delphi
function TBlogApplication.Login(const LogonName, PlainPassword: RawUTF8): TMVCAction;
var
  Author: TSQLAuthor;
  SessionInfo: TCookieData;
begin
  if CurrentSession.CheckAndRetrieve<>0 then begin
    GotoError(result,HTTP_BADREQUEST);
    exit;
  end;
  Author := TSQLAuthor.Create(RestModel,'LogonName=?',[LogonName]);
  try
    if (Author.ID<>0) and Author.CheckPlainPassword(PlainPassword) then begin
      SessionInfo.AuthorName := Author.LogonName;
      SessionInfo.AuthorID := Author.ID;
      SessionInfo.AuthorRights := Author.Rights;
      CurrentSession.Initialize(@SessionInfo,TypeInfo(TCookieData));
      GotoDefault(result);
    end else
      GotoError(result,sErrorInvalidLogin);
  finally
    Author.Free;
  end;
end;
```

As you can see, this Login() method will be triggered from http://localhost:8092/blog/login with LogonName=...&plainpassword=... parameters. It will first check that there is no current session, retrieve the ORM Author corresponding to the LogonName, check the supplied password, and set the
SessionInfo: TCookieData structure with the needed information. A call to CurrentSession.Initialize() will compute the cookie, then prepare to send it to the client browser.

The Login() method returns a TMVCAction structure. As a consequence, the call to GotoDefault(result) will let the TMVCApplication processor render the Default() method, as if the /blog/default URI will have been requested. On invalid credential, an error page is displayed instead.

When a web page is computed, the following overridden method will be executed:

```pascal
function TBlogApplication.GetViewInfo(MethodIndex: integer): variant;
begin
  result := inherited GetViewInfo(MethodIndex);
  ObjAddProps(['blog', fBlogMainInfo, 'session', CurrentSession.CheckAndRetrieveInfo(TypeInfo(TCookieData))], result);
end;
```

It will append the session information from the cookie to the returned View data context, as such:

```json
{
  "Scope": {
    "articles": ... 
  },
  "main": {
    "pageTitle": "Default",
    "blog": {
      "Title": "mORMot BLOG",
      ... 
    },
    "session": {
      "AuthorName": "synopse",
      "AuthorID": 1,
      "AuthorRights": {
        "Comment": true,
        "Post": true,
        "Delete": true,
        "Administrate": true
      },
      "id": 1
    }
  }
}
```

Here, the session object will contain the TCookieData information, ready to be processed by the Mustache View - e.g. as session.AuthorName. In addition, your view may include some buttons for logged-only features, like comments or content edition, using boolean fields defined in session.AuthorRights.

For security reasons, before actually performing an action requiring a specific right, it is preferred to check from the Model if the user is effectively allowed. An attacker may have forged a fake cookie - even if it is very unlikely, since cookies are encrypted and signed. It is a good approach to treat all cookies information as an unsafe cache, acceptable for most operation, but which should always be dual-checked.

So your server code will call CurrentSession.CheckAndRetrieve then access the data RestModel for verification before any sensitive action is performed. Defining a common method could be handy:

```pascal
function TBlogApplication.GetLoggedAuthorID(Right: TSQLAuthorRight; ContentToFillAuthor: TSQLContent): TID;
var
  SessionInfo: TCookieData;
  author: TSQLAuthor;
begin
```

```pascal
SAD - mORMot Framework - Rev. 1.18
Page 533 of 2552
```
```pascal
result := 0;
if (CurrentSession.CheckAndRetrieve(@SessionInfo,TypeInfo(TCookieData))>0) and 
(Right in SessionInfo.AuthorRights) then
  with TSQLAuthor.AutoFree(author,RestModel,SessionInfo.AuthorID) do
    if Right in author.Rights then begin
      result := SessionInfo.AuthorID;
      if ContentToFillAuthor<>nil then begin
        ContentToFillAuthor.Author := pointer(result);
        ContentToFillAuthor.AuthorName := author.LogonName;
        end;
      end;
    end;
end;

It will be used as such, e.g. to verify if a user can comment an article:

```pascal
function TBlogApplication.ArticleComment(ID: TID;
  const Title,Comment: RawUTF8): TMVCAction;
var
  comm: TSQLComment;
  AuthorID: TID;
  error: string;
begin
  with TSQLComment.AutoFree(comm) do begin
    AuthorID := GetLoggedAuthorID(canComment,comm);
    if AuthorID=0 then begin
      GotoError(result,sErrorNeedValidAuthorSession);
      exit;
    end;
    ...
  end;
end;

Eventually, when the browser asks for the /blog/logout URI, the following method will be executed:

```pascal
function TBlogApplication.Logout: TMVCAction;
begin
  CurrentSession.Finalize;
  GotoDefault(result);
end;

The session cookie will then be deleted on the browser side.

Note that if any deprecated or invalid cookie is detected by the mORMot MVC server, it will also be automatically deleted on the browser side.

### 19.3. Writing the Views

See Mustache template engine (page 505) for a description of how rendering take place in this MVC/MVVM application. You will find the Mustache templates in the "Views" sub-folder of the "30 - MVC Server" sample application.

You will find some *.html files, one per command expecting a View, and some *.partial files, which are some kind of re-usable sub-templates - we use them to easily compute the page header and footer, and to have a convenient way of gathering some piece of template code, to be re-used in several *.html views.

Here is how Default.html is defined:

```html
{{header}}
{{masthead}}
  <div class="blog-header">
    <h1 class="blog-title">{{main.blog.title}}</h1>
    <p class="lead blog-description">{{main.blog.description}}</p>
  </div>
  <div class="row">
    <div class="col-sm-8 blog-main">
      {{#Scope}}
      {{articlerow}}
    </div>
```
The {{>partials}} are easily identified, as other {{...}} value tags. The main partial is {{>articlerow}}, which will display all articles list.

{{WikiToHtml main.blog.about}} is an Expression Block able to render some simple text into proper HTML, using a simple Wiki syntax.

{{MonthToText PublishedMonth}} will execute a custom Expression Block, defined in our TBlogApplication, which will convert the obfuscated TSQLArticle.PublishedMonth integer value into the corresponding name and year:

```pascal
procedure TBlogApplication.MonthToText(const Value: variant;
on result: variant);
const MONTHS: array[0..11] of string = ('January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December');
var month: integer;
beginn  if VariantToInteger(Value, month) and (month > 0) then
   result := MONTHS[month mod 12] + ' ' + IntToStr(month div 12);
end;
```

The page displaying the Author information is in fact quite simple:

```html
{{header}}
{{masthead}}
  <div class="blog-header">User {{Author.LogonName}}
</div>
  <div class="lead blog-description">{{Author.FirstName}} {{Author.FamilyName}}
</div>
  <div class="panel panel-default">
    <div class="panel-heading">Information about <strong>{{Author.LogonName}}</strong></div>
    <div class="panel-body">
      {{TSQLAuthor.HtmlTable Author}}
    </div>
  </div>
{{articlerow}}
{{footer}}

It will share the same {{>partials}}, for a consistent and maintainable web site design, but in fact most of the process will take place by the magic of two tags:

- {{TSQLAuthor.HtmlTable Author}} is an Expression Block linked to TMVCApplication.RestModel ORM, which will create a HTML table - with the syntax expected by our Bootstrap CSS - for a TSQLAuthor record, identifying the property types and display them as expected (e.g. for dates or time stamps, or for enumerates or sets).
- `{{>articlerow}}` is a partial also shared with `ArticleView.html`, which will render a list of TSQLArticle encoded as `{{#Articles}}...{{/Articles}}` sections.

Take a look at the `mORMotMVC.pas` unit: you will discover that every aspect of the MVC process has been divided into small classes, so that the framework is able to create web applications, but also any kind of MVC applications, including mobile or VCL/FMX apps, and/or reporting - using `mORMotReport.pas`. 
20. Hosting

We could identify several implementation patterns of a mORMot server and its clients:
- Stand-alone application, either in the same process or locally on the same computer;
- Private self-hosting, e.g. in a corporate network, with a mORMot executable or service publishing some content to clients locally or over the Internet (directly from a DMZ or via a VPN);
- Cloud hosting, using a dedicated server in a data-center, or any cloud solution based on virtualization;
- Mixed hosting, using CDN network services to cache most of the requests of your mORMot server.

As we already stated, our Client-Server process (page 318) allow all these patterns. We will now detail some hosting schemes.
20.1. Windows and Linux hosted

The current version of the framework fully supports deploying the mORMot servers on the Windows platform, either as a Win32 executable, or - for latest versions of the Delphi compiler - as a Win64 executable.

Linux support (via FPC 3.2.x) is available, but we face some FPC compiler-level issue with FPC 2.x, which does not supply the needed interface RTTI - see http://bugs.freepascal.org/view.php?id=26774.. - so that the SOA and MVC features are not working directly non old FPC revision, so you need to generate the RTTI from a Delphi compiler, as stated below (page 655). For the client side, there is no limitation, thanks to our Cross-Platform clients (page 481), which is perfectly supported even by oldest FPC compiler under Linux. The Linux backend available in latest Delphi is not supported, since FPC 2.x gives pretty good results (we use it on production since years), and a Delphi Enterprise licence is required to access it - which we don’t have.

In practice, a mORMot server expects much lower hardware requirements (in CPU, storage and RAM terms) than a regular IIS-WCF-MSSQL-.Net stack. And it requires almost no maintenance.

As a consequence, the potential implementation schemes could be hosted as such:

- Stand-alone application, without any explicit server;
- Self-hosted service running on the corporate file server, or on a small dedicated VM or recycled computer (for best performance, just put your data on a new SSD on the old hardware PC);
- Cloud services running Windows Server, with minimal configuration: IIS, .Net or MS SQL are not necessary at all - a cheap virtual system with 512 MB of memory is enough to run your mORMot service and serve hundreds of clients;
- Linux servers, with no dependency (even latest version of SQLite3 is statically linked to the executables), using less hardware resource.

In the cloud, since every resource used is monitored and billed, you would like to minimize RAM use: you should better take a look at http://www.delphitools.info/2013/11/20/moving-hosts-now-settled.. and http://www.delphitools.info/2013/11/29/flush-windows-file-cache.. for practical advices and feedbacks.

About the edition of Windows to be used, of course IT people will ensure you that Windows Server is mandatory. But from our tests, you will obtain pretty good results, even with a regular Windows 7 or 8 version of the operating system. On the other side, it is not serious to envisage hosting a server on Windows XP, which is not supported any more by Microsoft - even if technically a mORMot server will work very well on this deprecated platform.

Of course, if you use External SQL database access (page 239), the hardware and hosting expectations may vary. It will depend on the database back-end used, and will necessarily be much more demanding than our internal SQLite3 database engine. In practice, a mORMot server using a SQLite3 engine running on a SSD hardware, in lmExclusive mode - see ACID and speed (page 222) - runs faster than most SQL or NoSQL engines available, since it will be hosted within the mORMot server process itself - see Highly concurrent clients performance (page 338).

20.2. Deployment Architecture

About hosting architecture, the easiest is to have your main TSQLRestServer class handling the service, in conjunction with other Client-Server process (like ORM). See General mORMot architecture - Client / Server (page 74) about this generic Client-Server architecture and the "Shared server" next paragraph.
But you may find out some (good?) reasons which main induce another design:

- For better scalability, you should want to use a dedicated process (or even dedicated hardware) to split the database and the service process;
- For security reasons, you want to expose only services to your Internet clients, and setup a DMZ hosting only services, and separate database with logic instance;
- Services are not the main part of your business, and you would like to enable or disable the published SOA scope, on demand;
- To implement an efficient solution for the most complex kind of application, as provided by Domain-Driven Design (page 99);
- Your main data will be hosted on high performance SSD / NAS drives with safe RAID, but some data should better be hosted on cheaper storage (e.g. Audit Trail for change tracking (page 178));
- You are selling one product, to be run on several environments (debugging / production, starter / corporate editions, centralized / P2P design...), depending on your clients demand;
- Whatever your IT people or managers want mORMot to.

Also consider per-table redirection - see Redirect to an external TSQLRest (page 233), or Master/slave replication (page 180), for even more advanced hosting abilities. See for instance the Corporate Servers Redirection (page 235) and GG diagrams.

The possibilities are endless, so we will here below only present some typical use-cases.

### 20.2.1. Shared server

This is the easiest configuration: one HTTP server instance, which serves both ORM and Services. On practice, this is perfectly working and scalable.

![Diagram of Shared Server](image)

*Service Hosting on mORMot - shared server*

You can tune this solution, as such:
- Setting the group user rights properly - see below (page 546) - you can disable the remote ORM access from the Internet, for the AJAX Clients - but allow rich Delphi clients (like PC1) to access the ORM;
- You can have direct in-process access to the service interfaces from the ORM, and vice-versa: if your services and ORM are deeply inter-dependent, direct access will be the faster solution.
20.2.2. Two servers

In this configuration, two physical servers are available:

- A network DMZ is opened to serve only service content over the Internet, via "HTTP server 2";
- Then on the local network, "HTTP server 1" is used by both PC 1 and Services to access the ORM;
- Both "PC Client 1" and the ORM core are able to connect to Services via a dedicated "HTTP server 3".

Service Hosting on mORMot - two servers

Of course, the database will be located on "PC Server internal", i.e. the one hosting the ORM, and the Services will be one regular client: so we may use CRUD level cache (page 359) on purpose to enhance performance. In order to access the remote ORM features, and provide a communication endpoint to the embedded services, a TSQLRestServerRemoteDB kind of server class can be used.
20.2.3. Two instances on the same server

This is the most complex configuration. In this case, only one physical server is deployed:
- A dedicated "HTTP server 2" instance will serve service content over the Internet (via a DMZ configuration of the associated network card);
- "PC Client 1" will access to the ORM via "HTTP server 1", or to services via "HTTP server 3";
- For performance reasons, since ORM and Services are on the same computer, using named pipes (or even local Windows Messages) instead of slower HTTP-TCP/IP is a good idea: in such case, ORM will access services via "Named Pipe server 2", whereas Services will serve their content to the ORM via "Named Pipe server 1".

Service Hosting on mORMot - one server, two instances

Of course, you can make any combination of the protocols and servers, to tune hosting for a particular purpose. You can even create several ORM servers or Services servers (grouped per features family or per product), which will cooperate for better scaling and performance.

If you consider implementing a stand-alone application for hosting your services, and has therefore basic ORM needs (e.g. you may need only CRUD statements for handling authentication), you may use the lighter TSQLRestServerFullMemory kind of server instead of a full TSQLRestServerDB, which will embed a SQLite3 database engine, perhaps not worth it in this case.
20.2.4. Scaling via CDN

Our beloved stateless REST (page 311) model, in conjunction with *Browser speed-up for unmodified requests* (page 379) will enable several levels of caching, from a local proxy cache - see e.g. http://www.squid-cache.org.. or http://www.varnish-cache.org.. - or an external Content Delivery Network (CDN) service - e.g. http://www.cloudflare.com..

Your mORMot server may be able to publish some dynamic HTML pages, or simple generic JSON services, and then let the CDN do the caching. An expiration time out of 30 seconds, configured at CDN level, will definitively help your web application to scale to thousands of visitors.

In practice, static content - see *Returning file content* (page 379) - or some simple JSON requests - returned via Ctxt.Results() or an interface-based service - will benefit of using such a CDN.

When any client requests the mORMot server URI, it will be in fact redirected to the closest CDN node available. For instance, some client in Canada will be redirected to the "CDN US" server, or one mobile client in France will be redirected to the "CDN UK" server.

Then each CDN will check if the requested URI is already in its cache, according to its settings, and the expiration parameters which may be set within the HTTP headers of the cache header. If the resource is in local cache, it will be returned to the client immediately. If the resource is not in its cache, the CDN node will ask the mORMot server, cache the returned content, then return this content to the client. Any further attempt to this URI, compatible with the expiration parameters, won’t trigger any request to the mORMot server.

Of course, you can define some URI patterns to never be cached, and point directly to the mORMot server. All authenticated services, for instance will need direct access to the mORMot server, since the below (page 552) will append a session-private signature to each URI. Just ensure that you disabled authentication - using TSQLRestServer.ServiceMethodByPassAuthentication() for method-based services, or TServiceFactoryServer.ByPassAuthentication property for interface-based services.
The per-session signature appended at each URI will indeed void any attempt of third-party cache.

If your project starts to have success, using a CDN is an easy and cheap way of increasing your number of clients. Your mORMot server will focus on its own purpose, which may be safe storage, authentication and high-level SOA, then let the remaining content be served by such a third-party caching system.
21. Security

The framework tries to implement security via:
  - Process safety;
  - Authentication;
  - Authorization.

Process safety is implemented at every Multi-tier architecture (page 88) level:
  - Strong encryption to keep information private - see AES encryption over HTTP (page 333) and below (page 571);
  - Atomicity of the SQLite3 database - see ACID and speed (page 222);
  - Stateless ORM (page 314) architecture to avoid most synchronization issues;
  - Object-Relational Mapping (ORM) (page 92) associated to the Object pascal strong type syntax;
  - Extended test coverage - see below (page 626) - of the framework core.

Authentication allows user identification:
  - Build-in optional authentication mechanism, implementing both per-user sessions and individual REST Query Authentication;
  - Authentication groups are used for proper authorization;
  - Several authentication schemes, from very secure HMAC-SHA256 based challenging to weak but simple authentication;
  - Class-based architecture, allowing custom extension.

Authorization of a given process is based on the group policy, after proper authentication:
  - Per-table access right functionalities built-in at lowest level of the framework;
  - Per-method execution policy for interface-based services;
- General high-level security attributes, for SQL or Service remote execution.

Process safety has already been documented (see links above).

We will now give general information about both authentication and authorization in the framework.
21.1. Authentication

Extracted from Wikipedia:

Authentication (from Greek: "real" or "genuine", from "author") is the act of confirming the truth of an attribute of a datum or entity. This might involve confirming the identity of a person or software program, tracing the origins of an artifact, or ensuring that a product is what its packaging and labeling claims to be. Authentication often involves verifying the validity of at least one form of identification.

21.1.1. Principles

How to handle authentication in a RESTful Client-Server architecture is a matter of debate.

Commonly, it can be achieved, in the SOA over HTTP world via:
- **HTTP basic auth** over HTTPS;
- **Cookies** and session management;
- **Query Authentication** with additional signature parameters.

We’ll have to adapt, or even better mix those techniques, to match our framework architecture at best.

Each authentication scheme has its own PROs and CONs, depending on the purpose of your security policy and software architecture:

<table>
<thead>
<tr>
<th>Criteria</th>
<th>HTTPS basic auth</th>
<th>Cookies+Session</th>
<th>Query Auth.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browser integration</td>
<td>Native</td>
<td>Native</td>
<td>Via JavaScript</td>
</tr>
<tr>
<td>User Interaction</td>
<td>Rude</td>
<td>Custom</td>
<td>Custom</td>
</tr>
<tr>
<td>Web Service use (rough estimation)</td>
<td>95%</td>
<td>4%</td>
<td>1%</td>
</tr>
<tr>
<td>Session handling</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Session managed by</td>
<td>Client</td>
<td>Server</td>
<td>N/A</td>
</tr>
<tr>
<td>Password on Server</td>
<td>Yes</td>
<td>Yes/No</td>
<td>N/A</td>
</tr>
<tr>
<td>Truly Stateless</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Truly RESTful</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>HTTP-free</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

21.1.1.1. HTTP basic auth over HTTPS

This first solution, based on the standard HTTPS protocol, is used by most web services. It’s easy to implement, available by default on all browsers, but has some known draw-backs, like the awful authentication window displayed on the Browser, which will persist (there is no LogOut-like feature here), some server-side additional CPU consumption, and the fact that the user-name and password are transmitted (over HTTPS) into the Server (it should be more secure to let the password stay only on the client side, during keyboard entry, and be stored as secure hash on the Server).
The supplied TSQLHttpClientWinHTTP and TSQLHttpClientWinINet clients classes are able to connect using HTTPS, and the THttpApiServer server class can send compatible content.

### 21.1.1.2. Session via Cookies

To be honest, a session managed on the Server is not truly Stateless. One possibility could be to maintain all data within the cookie content. And, by design, the cookie is handled on the Server side (Client in fact don't even try to interpret this cookie data: it just hands it back to the server on each successive request). But this cookie data is application state data, so the client should manage it, not the server, in a pure Stateless world.

The cookie technique itself is HTTP-linked, so it's not truly RESTful, which should be protocol-independent. Since our framework does not provide only HTTP protocol, but offers other ways of transmission, Cookies were left at the baker's home.

### 21.1.1.3. Query Authentication

**Query Authentication** consists in signing each RESTful request via some additional parameters on the URI. See [http://broadcast.oreilly.com/2009/12/principles-for-standardized-rest-authentication.html](http://broadcast.oreilly.com/2009/12/principles-for-standardized-rest-authentication.html) about this technique. It was defined as such in this article:

*All REST queries must be authenticated by signing the query parameters sorted in lower-case, alphabetical order using the private credential as the signing token. Signing should occur before URI encoding the query string.*

For instance, here is a generic URI sample from the link above:

```
GET /object?apiKey=Qwerty2010
```

should be transmitted as such:

```
GET /object?timestamp=1261496500&apiKey=Qwerty2010&signature=abcdef0123456789
```

The string being signed is "/object?apikey=Qwerty2010&timestamp=1261496500" and the signature is the SHA256 hash of that string using the private component of the API key.

This technique is perhaps the more compatible with a Stateless architecture, and can also been implemented with a light session management.

Server-side data caching is always available. In our framework, we cache the responses at the SQL level, not at the URI level (thanks to our optimized implementation of GetJSONObjectAsSQL, the URI to SQL conversion is very fast). So adding this extra parameter doesn’t break the cache mechanism.

### 21.1.2. Framework authentication

Even if, theoretically speaking, **Query Authentication** sounds to be the better for implementing a truly RESTful architecture, our framework tries to implement a Client-Server design.

In practice, we may consider two way of using it:
- With no authentication nor user right management (e.g. for local access of data, or framework use over a secured network);
- With per-user authentication and right management via defined *security groups*, and a per-query authentication, following several protocols (a set of *mORMot* flavors, Windows NTLM/Kerberos, or any custom scheme).

According to RESTful principle, handling per-session data is not to be implemented in such
architecture. A minimal "session-like" feature was introduced only to handle user authentication with very low overhead on both Client and Server side. The main technique used for our security is therefore *Query Authentication*, i.e. a per-URI signature.

If the `aHandleUserAuthentication` parameter is left to its default false value for the `TSQLRestServer` Create constructor, no authentication is performed. All tables will be accessible by any client, as stated in below (page 559). As stated above, for security reasons, the ability to execute INSERT / UPDATE / DELETE SQL statement via a RESTful POST command is never allowed by default with remote connections: only SELECT can be executed via this POST verb.

If authentication is enabled for the Client-Server process (i.e. if the `aHandleUserAuthentication` parameter is set to true at the `TSQLRestServer` instance construction), the following security features will be added:

- On the Server side, a dedicated service, accessible via the `ModelRoot/Auth` URI is to be called to register an User, and create an in-memory session;
- Client *should* open a session to access to the Server, and after authentication validation (e.g. via Username / Password pair, or Windows credentials);
- Each CRUD statement is checked against the authenticated User security group, via the `AccessRights` column and its GET / POST / PUT / DELETE per-table bit sets;
- Thanks to *Per-User* authentication, any SQL statement commands may be available via the RESTful POST verb for an user with its `AccessRights` group field containing a reSQL flag in its `AllowRemoteExecute`;
- Each REST request will expect an additional parameter, named `session_signature`, to every URL. Using the URI instead of cookies allows the signature process to work with all communication protocols, not only HTTP;
- Of course, you have the opportunity to tune or even by-pass the security for a given service (method-based or interface-based), on need: e.g. to allow some methods only to your system administrators, or to serve public HTML content.

### 21.1.2.1. *Per-User* authentication

On the Server side, two tables, defined by the `TSQLAuthGroup` and `TSQLAuthUser` classes, will handle respectively per-group access rights (authorization), and user validation (authentication). In the database, they will be persisted as `AuthGroup` and `AuthUser` tables.

The Server will search for any class inheriting from `TSQLAuthGroup` and `TSQLAuthUser` in its Model. By default, you can use plain `TSQLAuthGroup` and `TSQLAuthUser` classes - and if none is defined in the model, and authentication is enabled, those mandatory classes will be added. But you can inherit from `TSQLAuthGroup` and `TSQLAuthUser`, and define e.g. your own fields, for any custom purpose at *Group* or *User* level. The exact class types are available from the `TSQLRestServer` `SQLAuthUserClass` and `SQLAuthGroupClass` properties.

Since the whole records will be loaded and persisted in memory at every authentication, do not store too much data in those tables: for instance, do not put historical data (like previous client activity), or huge BLOBs (like detailed pictures) - a dedicated table or set of tables will be a better idea.

Here is the layout of the default `AuthGroup` table, as defined by the `TSQLAuthGroup` class type:

| ID   | TID   | AccessRights : RawUTF8 | Ident : RawUTF8 | SessionTimeout : integer |
**AuthGroup Record Layout**

The AccessRights column is a textual CSV serialization of the TSQLAccessRights record content, as expected by the TSQLRestServer.URI method. Using a CSV serialization, instead of a binary serialization, will allow the change of the MAX_SQLTABLES constant value.

The AuthUser table, as defined by the TSQLAuthUser class type, is defined as such:

<table>
<thead>
<tr>
<th>ID</th>
<th>TID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>TSQLRawBlob</td>
</tr>
<tr>
<td>DisplayName</td>
<td>RawUTF8</td>
</tr>
<tr>
<td>GroupRights</td>
<td>TSQLAuthGroup</td>
</tr>
<tr>
<td>LogonName</td>
<td>RawUTF8</td>
</tr>
<tr>
<td>PasswordHashHexa</td>
<td>RawUTF8</td>
</tr>
</tbody>
</table>

**AuthUser Record Layout**

Each user has therefore its own associated AuthGroup table, a name to be entered at login, a name to be displayed on screen or reports, and a SHA256 hash of its registered password (with optional PBKDF2_HMAC_SHA256 derivation). A custom Data BLOB field is specified for your own application use, but not accessed by the framework.

By default, the following security groups are created on a void database:

<table>
<thead>
<tr>
<th>Group</th>
<th>POST SQL</th>
<th>SELECT SQL</th>
<th>Auth R</th>
<th>Auth W</th>
<th>Tables R</th>
<th>Tables W</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admin</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>User</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Guest</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

'Admin' will be the only able to execute remote not SELECT SQL statements for POST commands (reSQL flag in TSQLAccessRights. AllowRemoteExecute) and modify the Auth* tables (i.e. AuthUser and AuthGroup).

'Admin' and 'Supervisor' will allow any SELECT SQL statements to be executed, even if the table can't be retrieved and checked (corresponding to the reSQLSelectWithoutTable flag).

'User' won't have the reSQLSelectWithoutTable flag, nor the right to retrieve the Auth* tables data for other users.

'Guest' won't have access to the interface-based remote JSON-RPC service (no reService flag), nor perform any modification to a table: in short, this is an ORM read-only limited user.

Please see below (page 559) and the TSQLAccessRights documentation for all available options and use cases.

Then the corresponding 'Admin', 'Supervisor' and 'User' AuthUser accounts are created, with the default 'synopse' password.

You MUST override those default 'synopse' passwords for each AuthUser record to a custom genuine value.

A typical JSON representation of the default security user/group definitions are the following:

```json
[{
    "AuthUser": [ 
      {"RowID":1, "LogonName":"Admin", "DisplayName":"Admin",
```
"PasswordHashHexa":"67aeea294e1cb51236fd7829c55ec820ef888e8e221814d24d83b3dc4d825dd",
"GroupRights":1, "Data":null},
{RowID:2, "LogonName":"Supervisor", "DisplayName":"Supervisor",
"PasswordHashHexa":"67aeea294e1cb51236fd7829c55ec820ef888e8e221814d24d83b3dc4d825dd",
"GroupRights":2, "Data":null},
{RowID:3, "LogonName":"User", "DisplayName":"User",
"PasswordHashHexa":"67aeea294e1cb51236fd7829c55ec820ef888e8e221814d24d83b3dc4d825dd",
"GroupRights":3, "Data":null}]

Of course, you can change AuthUser and AuthGroup table content, to match your security requirements, and application specifications. You can specify a per-table CRUD access, via the AccessRights column, as we stated above, speaking about the TSQLAccessRights record layout.

This will implement both Query Authentication together with a group-defined per-user right management.

21.1.2.2. Session handling

A dedicated RESTful service, available from the ModelRoot/Auth URI, is to be used for user authentication, handling so called sessions.

In mORMot, a very light in-memory set of sessions is implemented:
- The unique ModelRoot/Auth URI end-point will create a session after proper authorization;
- In-memory session allows very fast process and reactivity, on Server side;
- Sessions could be optionally persisted on disk at server shutdown, to avoid breaking existing client connections;
- An integer session identifier is used for all authorization process, independently from the underlying authentication scheme (i.e. mORMot is not tied to cookies, and its session process is much more generic).

Those sessions are in-memory TAuthSession class instances. Note that this class does not inherit from a TSQLRecord table so won't be remotely accessible, for performance and security reasons.

The server methods should not have to access those TAuthSession instances directly, but rely on the SessionID identifier. But you can still access the current session properties, e.g. the remote user, thanks to methods like TSQLRestServer.SessionGetUser(): TSQLAuthUser.

When the Client is about to close (typically in TSQLRestClientURI.Destroy), a GET ModelRoot/auth?UserName=...&Session=... request is sent to the remote server, in order to explicitly close the corresponding session in the server memory (avoiding most re-play attacks).

Note that each opened session has an internal TimeOut parameter (retrieved from the associated TSQLAuthGroup table content): after some time of inactivity, sessions are closed on the Server Side.

In addition, sessions are used to manage safe cross-client transactions:
- When a transaction is initiated by a client, it will store the corresponding client Session ID, and use it to allow client-safe writing;
- Any further write to the DB (Add/Update/Delete) will be accessible only from this Session ID, until the transaction is released (via commit or rollback);
- If a transaction began and another client session try to write on the DB, it will wait until the current
transaction is released - a timeout may occur if the server is not able to acquire the write status within some time;
- This global write locking is defined by the TSQLRest.AcquireWriteMode / AcquireWriteTimeOut properties, and used on the Server-Side by TSQLRestServer.URI - you can change this behavior by setting e.g. AcquireWriteMode := amBackgroundThread which will lock any write process to be executed in a dedicated thread: this may be mandatory is your database client expects the transaction process to take place in the same thread (e.g. MS SQL);
- If the server do not handle Session/Authentication, transactions can be unsafe, in a multi-client concurrent architecture.

For performance reasons in a multi-client environment, it's mandatory to release a transaction (via commit or rollback) as soon as possible, using e.g. BATCH sequences for adding/updating/deleting records (page 350), or - even better - write dedicated Client-Server services via interfaces (page 419) which will process the whole transaction in one step, following the Unit Of Work pattern (page 353).

You can specify an optional file name parameter to the TSQLRestServer.Shutdown() method, which will save the current server state into a local file. Then, if you restart the server in a short time, you may be able to restore all session information by using TSQLRestServer.SessionsLoadFromFile(). This feature will enable e.g. a quick and transparent ORM Server executable upgrade, in production. But note that even if sessions are persisted and able to be restored, any session-dependent complex data - like server-side temporary Instances life time implementation (page 429) as generated by interface-based services - won't be available. This session temporary backup/restore will make sense only when the server is in ORM mode, not when used as SOA.

21.1.3. Authentication schemes

21.1.3.1. Class-driven authentication

Authentication is implemented in mORMot via the following classes:

![TSQLRestServerAuthentication classes hierarchy]

In fact, you can use one of the following RESTful authentication schemes:

<table>
<thead>
<tr>
<th>class</th>
<th>Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLRestServerAuthenticationDefault</td>
<td>mORMot secure authentication, as a proprietary dual-pass challenge and SHA256/PBKDF2_HMAC_SHA256 hashing</td>
</tr>
<tr>
<td>TSQLRestServerAuthenticationSSPI</td>
<td>Windows authentication, via the logged user</td>
</tr>
</tbody>
</table>
All those classes will identify a TSQLAuthUser record from a user name. The associated TSQLAuthGroup is then used later for authorization.

You can add your own custom authentication scheme by defining your own class, inheriting from TSQLRestServerAuthentication.

By default, no authentication is performed.

If you set the aHandleUserAuthentication parameter to true when calling the constructor TSQLRestServer.Create(), both default secure mORMot authentication and Windows authentication are available. In fact, the constructor executes the following:

```pascal
constructor TSQLRestServer.Create(aModel: TSQLModel; aHandleUserAuthentication: boolean);
( ...)
if aHandleUserAuthentication then
  // default mORMot authentication schemes
  AuthenticationRegister([TSQLRestServerAuthenticationDefault,TSQLRestServerAuthenticationSSPI]);
( ...)
```

In order to define one or several authentication scheme, you can call the AuthenticationRegister() and AuthenticationUnregister() methods of TSQLRestServer.

### 21.1.3.2. mORMot secure RESTful authentication

The TSQLRestServerAuthenticationDefault class implements a proprietary but secure RESTful Authentication (page 546).

As stated in Enhanced sample: remote SQL access (page 439), typical client-side authentication is performed using the following command:

```pascal
MyClient.SetUser('User', 'synopse'); // default user for Security tests
```

Here are the typical steps to be followed in order to create a new user session via mORMot authentication scheme:
- Client sends a GET ModelRoot/auth?UserName=... request to the remote server - with the above command, it will be GET ModelRoot/auth?UserName=User;
- Server answers with a hexadecimal nonce contents (valid for about 5 minutes), encoded as JSON result object;
- Client sends a GET ModelRoot/auth?UserName=...&PassWord=...&ClientNonce=... request to the remote server, in which ClientNonce is a random value used as Client nonce, and Password is computed from the log-on and password entered by the User, using both Server and Client nonce as salt;
- Server checks that the transmitted password is valid, i.e. that its matches the hashed password stored in its database and a time-valid Server nonce - if the value is not correct, authentication fails;
- On success, Server will create a new in-memory session and return the session number and a private key to be used during the session (encoded as JSON result object);
- On any further access to the Server, a &session_signature= parameter is added to the URL, and will be checked against the valid sessions in order to validate the request.

Query Authentication is handled at the Client side in TSQLRestClientURI.SessionSign method, by
computing the session_signature parameter for a given URL, according to the TSQLRestServerAuthentication class used.

In order to enhance security, the session_signature parameter will contain, encoded as 3 hexadecimal 32-bit cardinals:
- The Session ID (to retrieve the private key used for the signature);
- A Client Time Stamp (in 256 ms resolution) which must be greater or equal than the previous time stamp received;
- The URI signature, using the session private key, the user hashed password, and the supplied Client Time Stamp as source for its crc32 hashing algorithm.

Such a classical 3 points signature will avoid most man-in-the-middle (MITM) or re-play attacks.

Here is typical signature to access the root URL

```
root?session_signature=0000004C000F6BE365D8D454
```

In this case, 0000004C is the Session ID, 000F6BE3 is the client time stamp (aka nonce), and 65D8D454 is the signature, computed by the following Delphi expression:

```
(crc32(crc32(fPrivateSaltHash,PTimestamp,8),pointer(aURL),aURLlength)=aSignature);
```

For instance, a RESTful GET of the TSQLRecordPeople table with RowID=6 will have the following URI:

```
root/People/6?session_signature=0000004C000F6D02E24541C
```

For better Server-side performance, the URI signature will use fast crc32 hashing method, and not the more secure (but much slower) SHA256. Since our security model is not officially validated as a standard method (there is no standard for per URI authentication of RESTful applications), the better security will be handled by encrypting the whole transmission channel, using standard HTTPS with certificates signed by a trusted CA, validated for both client and server side. The security involved by using crc32 will be enough for most common use. Note that the password hashing and the session opening will use SHA256 or PBKDF2_HMAC_SHA256, to enhance security with no performance penalty.

In our implementation, for better Server-side reaction, the session_signature parameter is appended at the end of the URI, and the URI parameters are not sorted alphabetically, as suggested by the reference article quoted above. This should not be a problem, either from a Delphi Client or from a AJAX / JavaScript client.

On practice, this scheme is secure and very fast, perfect for a Delphi client, or an AJAX application. If you expect a higher level of security for the URI signature, you may consider setting a cryptographic-level MD5/SHA1/SHA256/SHA512 hash, by selecting a given TSQLRestServerAuthenticationSignedURIAlgo on server side.

### 21.1.3.3. Authentication using Windows credentials

#### 21.1.3.3.1. Windows Authentication

By default, the hash of the user password is stored safely on the server side. This may be an issue for corporate applications, since a new user name / password pair is to be defined by each client, which may be annoying.

Since revision 1.18 of the framework, mORMot is able to use Windows Authentication to identify any user. That is, the user does not need to enter any name nor password, but her/his Windows credentials, as entered at Windows session startup, will be used.
If the SSPIAUTH conditional is defined (which is the default), any call to TSQLRestClientURI.SetUser() method with a void aUserName parameter will try to use current logged name and password to perform a secure Client-Server authentication. It will in fact call the class function TSQLRestServerAuthenticationSSPI.ClientSetUser() method.

In this case, the aPassword parameter will identify if either NTLM or Kerberos authentication scheme is to be used: it may contain the SPN domain name to enabled Kerberos - see next section. This will be transparent to the framework, and a regular session will be created on success.

Only prerequisite is that the TSQLAuthUser table shall contain a corresponding entry, with its LogonName column equals to 'DomainName' value. This data row won't be created automatically, since it is up to the application to allow or disallow access from an authenticated user: you can be member of the domain, but not eligible to the application.

### 21.1.3.3.2. Using NTLM or Kerberos

Kerberos is the preferred authentication protocol for Windows Server 2003 and subsequent Active Directory domains.

Kerberos authentication offers the following advantages over NTLM authentication:
- Mutual authentication.
  - When a client uses the Kerberos protocol for authentication with a particular service on a particular server, Kerberos provides the client with an assurance that the service is not being impersonated by malicious code on the network.
- Simplified trust management.
  - Networks with multiple domains no longer require a complex set of explicit, point-to-point trust relationships.
- Enhanced security.
  - The old NTLM protocol suffers from several weaknesses, which have been fixed by Kerberos.
- Performance.
  - Offers improved performance, mostly for server applications.

Requirements for Kerberos authentication are the following:
- Client and Server must join a domain, and the trusted third party must exist; if client and server are in different domain, these two domains must be configured as two-way trust.
- SPN must have been registered properly. Service Principal Name (SPNs) are unique identifiers for services running on servers. Each service that will use Kerberos authentication needs to have an SPN set for it so that clients can identify the service on the network. It is registered in Active Directory under either a computer account or a user account. See below for corresponding instructions.

Typical use case of either Kerberos or NTLM are defined by the aPassword parameter:
- Kerberos is used for a remote connection over a network and if aPassword is set to the expected SPN domain;
- NTLM is used over network connection if aPassword is empty;
- NTLM is used when making local connection.

Note that Kerberos is used only when making remote connection over a network; NTLM is used when making local connection.

To enable Kerberos authentication in mORMot, you need to register SPN for your service.

The format of an SPN is ServiceClass/Host:Port/ServiceName. Typically, SPN for your service,
developed with mORMot, looks like mymormotservice/myserver.mydomain.tld or http/myserver.mydomain.tld.

To list SPNs of a computer named MYSERVER, at the command prompt, type:

```bash
setspn -l myserver
```

Typically, you can see the following output:

```
Registered ServicePrincipalNames for CN=MYSERVER,OU=Computers,DC=domain,DC=tld:
    HOST/MYSERVER.domain.tld
    HOST/MYSERVER
```

If your service runs under SYSTEM or Network Service machine accounts, you can test Kerberos authentication by setting the `aPassword` parameter to value `HOST/MYSERVER.domain.tld` in the client code and run the application.

To register SPN for your service, at the command prompt, type:

- If your service run under SYSTEM or Network Service machine accounts:

  ```bash
  setspn -a mymormotservice/myserver.mydomain.tld myserver
  ```

- If your service run under another domain account:

  ```bash
  setspn -a mymormotservice/myserver.mydomain.tld myserviceaccount
  ```

Membership in `Domain Admins` group, or equivalent, is the minimum required to complete this procedure.


After registration, you can connect to the server as such:

```csharp
MyClient.SetUser('','mymormotservice/myserver.mydomain.tld'); // will use Kerberos
```

For good old NTLM, you can run:

```csharp
MyClient.SetUser('',''); // will use NTLM
```

Or directly call the `TSQLRestServerAuthenticationSSPI.ClientSetUser()` method.

The authentication mode used will appear in the log file, if you define `WITHLOG` conditional when building the service application and if `sllUserAuth` is in `TSQLLog.Family.Level` set.

Messages will be as follows:

```
NTLM Authentication success for domain\myuser
Kerberos Authentication success for domain\myuser
```

The framework authorization will then be processed as usual, for all features like RESTful ORM process and remote services.

### 21.1.3.4. Weak authentication

The `TSQLRestServerAuthenticationNone` class can be used if you trust your client (e.g. via a `https` connection). It will implement a weak but simple authentication scheme.

Here are the typical steps to be followed in order to create a new user session via this authentication scheme:

- Client sends a GET `ModelRoot/auth?UserName=...` request to the remote server;
- Server checks that the transmitted user name is valid, i.e. that it is available in the `TSQLAuthGroup` table - if the value is not correct, authentication fails
- On success, Server will create a new in-memory session and returns the associated session number
(encoded as decimal in the JSON result object);
- On any further access to the Server, a &session_signature= parameter is to be added to the URL with the correct session ID (encoded as hexadecimal), and will be checked against the valid sessions in order to validate the request.

For instance, a RESTful GET of the TSQLRecordPeople table with RowID=6 will have the following URI:

```
root/People/6?session_signature=0000004C
```

Here is some sample code about how to define this authentication scheme:

```
// on the Server side:
Server.AuthenticationRegister(TSQLRestServerAuthenticationNone);
...
// on the Client side:
TSQLRestServerAuthenticationNone.ClientSetUser(Client,'User');
```

The performance benefit is very small in comparison to TSQLRestServerAuthenticationDefault, so should not be used for Delphi clients.

### 21.1.3.5. HTTP Basic authentication

The Basic Authentication mechanism provides no confidentiality protection for the transmitted credentials. They are merely encoded with Base64 in transit, but not encrypted or hashed in any way. Basic Authentication is, therefore, typically used over HTTPS.

The TSQLRestServerAuthenticationHttpBasic class can be used to enable HTTP Basic authentication.

This class is not to be used with a mORMot client, since TSQLRestServerAuthenticationDefault provides a much better scheme, both safer and faster, but could be used in conjunction with some browser clients, over HTTPS.

### 21.1.4. Clients authentication

#### 21.1.4.1. Client interactivity

Note that with this design, it's up to the Client to react to an authentication error during any request, and ask again for the User pseudo and password at any time to create a new session. For multiple reasons (server restart, session timeout...) the session can be closed by the Server without previous notice.

In fact, the Client should just use create one instance of the TSQLRestClientURI classes as presented in JSON RESTful Client-Server (page 295), then call the SetUser method as such:

```
Check(Client.SetUser('User','synopse')); // use default user
```

Then an event handled can be associated to the TSQLRestClientURI. OnAuthenticationFailed property, in order to ask the user to enter its login name and password:

```
TOnAuthenticationFailed = function(Retry: integer;
var aUserName, aPassword: string): boolean;
```

Of course, if Windows Authentication is defined (see above), this event handler shall be adapted as expected. For instance, you may add a custom notification to register the corresponding user to the TSQLAuthUser table.

#### 21.1.4.2. Authentication using AJAX

Smart Mobile Studio support (page 486) can generate JavaScript code from its IDE. Our
template-based code generation make this solution perfectly integrated with our mORMot server, especially about authentication: you will find the same TSQLRestServerAuthenticationDefault and TSQLRestServerAuthenticationNone classes in our SynCrossPlatformREST.pas unit, ready to authenticate to the server.

In fact, there is also a command-line compiler available (named smsc.exe) which can create a .js file from SmartPascal code: you may use it to integrate the generated client to a regular HTML5 application (using e.g. Jquery or AngularJS).

Some stand-alone working JavaScript code has been published in our forum by a framework user (thanks, "RangerX"), which implements the authentication schema as detailed above. It uses jQuery, and HTML 5 LocalStorage, not cookies, for storing session information on the Client side.


### 21.1.5. JWT Authentication

As an alternative, you may use JSON Web Tokens (JWT) (page 379) for authentication.

On server side, you can assign a TJWTAbstract inherited instance to TSQLRestServer.JWTForUnauthenticatedRequest so that any client providing a valid JWT would be allowed to execute some requests.

```pascal
aJWTEngine := TJWTS3256.Create(aSecretKey, 60000, [jrcIssuer, jrcExpirationTime], [], aExpireMinutes);
aRestServer.JWTForUnauthenticatedRequest := aJWTEngine; // will be owned by aRestServer
```

On mORMot client side, you can provide a valid JWT via:

```pascal
aRestClientURI.SessionHttpHeader := AuthorizationBearer(aJWT);
```

Any kind of JSON/HTTPS client could easily connect to such a service, by providing a valid JWT as 'Authorization: Bearer ####' HTTP header. A dedicated authentication service may be used to return some JWT in exchange from some credential (typically a user-name/password) for your application.

In practice, for your internal MicroServices communication, you could therefore use regular mORMot secure RESTful authentication (page 552) over a WebSockets support (page 444), which are pretty stable and efficient. Then for a public API, you could use a regular TSQLHttpServer - see Network and Internet access via HTTP (page 325) - perhaps over a nginx reverse proxy (e.g. for Let's Encrypt HTTPS certification). Since the mORMot authentication is proprietary, using a JWT may sound more natural for a public API service, with a more relaxed JSON encoding and no contract. That is, on the server side, you define: ServiceDefine(...).ResultAsJSONObjectWithoutResult := true; and on the client side, you call the TSQLRestClientURI.ServiceDefineSharedAPI() method to follow a similar more standard JSON layout (i.e. JSON objects as input/output and not a JSON array without any contract negotiation).

By design, such clients won't be tied to any associated mORMot session or user. So Client-Server services via interfaces (page 419) should be only of sicSingle, sicShared or sicPerThread kind - see Instances life time implementation (page 429).

See also TSQLRestServer.JWTForUnauthenticatedRequestWhiteIP for additional security, to require the client to connect from a finite set of allowed IP addresses.
21.2. Authorization

By authorization, we mean the action to define an access policy to the RESTful resources, for an authenticated user. Even if this user may be a guest user (with no specific access credential), it should be identified as such, e.g. to serve public content.

The main principle is the principle of least privilege (also known as the principle of minimal privilege or the principle of least authority): in a particular abstraction layer of a computing environment, every module (such as a process, a user or a program depending on the subject) must be able to access only the information and resources that are necessary for its legitimate purpose.

It is most of the time implemented e.g. via Access Control Lists (ACL), set of capabilities or user groups. In mORMot, we defined user groups, associated to TSQLAuthGroup ORM class.

Today, authorization is part of a trust chain:
- In corporate networks, the Active Directory service gives a token for an already signed user, or LDAP allows access to resources;
- In social networks, protocols like OAuth allows to trust an user between services.

This allows the very convenient feature of single sign-on: the user can authenticate only once (e.g. at Windows logon), then he/she will be authenticated for its whole session, and each authorization will provide the appropriate rights. Our framework e.g. features NTLM / Kerberos authentication, as we just saw.

21.2.1. Per-table access rights

Even if authentication is disabled, a pointer to a TSQLAccessRights record, and its GET / POST / PUT / DELETE fields, is sent as a member of the parameter to the unique access point of the server class:

```pascal
procedure TSQLRestServer.URI(var Call: TSQLRestServerURIParams);
```

This will allow checking of access right for all CRUD operations, according to the table invoked. For instance, if the table TSQLRecordPeople has 2 as index in TSQLModel.Tables[], any incoming POST command for TSQLRecordPeople will be allowed only if the 2nd bit in RestAccessRights^.POST field is set, as such:

```pascal
case URI.Method of
  mPOST: begin // POST=ADD=INSERT
    if URI.Table=nil then begin
      (...) end
    else if not (URI.TableIndex in Call.RestAccessRights^.POST) then // check User
      Call.OutStatus := HTTP_FORBIDDEN else
    (...) end
```

Making access rights a parameter allows this method to be handled as pure stateless, thread-safe and session-free, from the bottom-most level of the framework.

On the other hand, the security policy defined by this global parameter does not allow tuned per-user authorization. In the current implementation, the SUPERVISOR_ACCESS_RIGHTS constant is transmitted for all handled communication protocols (direct access, Windows Messages, named pipe or HTTP). Only direct access via TSQLRestClientDB will use FULL_ACCESS_RIGHTS, i.e. will have AllowRemoteExecute parameter set to all possible flags.

The light session process, as implemented by Authentication (page 546), is used to override the access rights with the one defined in the TSQLAuthGroup.AccessRights field.
Be aware that this per-table access rights depend on the table order as defined in the associated TSQLModel. So if you add some tables to your database model, please take care to add the new tables after the existing. If you insert the new tables within current tables, you will need to update the access rights values.

21.2.2. Additional safety

A AllowRemoteExecute: TSQLAllowRemoteExecute field has been made available in the TSQLAccessRights record to tune remote execution, depending on the authenticated user, and the group he/she is part of.

This field adds some flags to tune the security policy, for both SQL or SOA dimensions.

21.2.2.1. SQL remote execution

In our RESTful implementation, the POST command with no table associated in the URI allows to execute any SQL statement directly. A GET command could also be used, either with the SQL statement transmitted as body (which is convenient, but not supported by all HTTP clients, since it is not standard), or inlined at URI level.

These special commands should be carefully tested before execution, since SQL misuses could lead into major security issues. Such execution on any remote connection, if the SQL statement is not a SELECT, is unsafe. In fact, if it may affect the data content.

By default, for security reasons, the AllowRemoteExecute field value in SUPERVISOR_ACCESS_RIGHTS constant does not include the reSQL flag. It means that no remote call will be allowed but for safe read-only SELECT statements.

When SELECT statements are sent, the server will always check for the table name specified in their FROM clause. If there is a single table appearing, its security policy will be checked against the GET[] flags of the corresponding table. If the SELECT statement is more complex (e.g. is a JOINed statement), then the reSQLSelectWithoutTable will be checked to ensure that the user has the right to execute such statements.

Another possibility of SQL remote execution is to add a sql=.... inline parameter to a GET request (with optional paging). The reUrlEncodedSQL flag is used to enable or disable this feature.

Last but not least, a WhereClause=... inline parameter can be added to a DELETE request. The reUrlEncodedDelete option is used to enable or disable this feature.

You can change the default safe policy by including or excluding reSQL, reSQLSelectWithoutTable, reUrlEncodedSQL or reUrlEncodedDelete flags in the TSQLAuthGroup. AccessRights. AllowRemoteExecute field of an authentication user session.

If security is a real concern, you should enable mORMot secure RESTful authentication (page 552) and URI signature on your server, so that only trusted clients may access to the server. This is the main security rule of the framework - in practice, those per table access right or SQL remote execution flags are more a design rule than a strong security feature. Since remote execution of any SQL statements can be unsafe, we recommend to write a dedicated server-side service (method-based or interface-based) to execute such statements instead, and disallow remote SQL execution; then clients can safely use those dedicated safe services, and/or ORM CRUD operations for simple data requests. It will also help your project to be not tied to SQL, so that a switch to a NoSQL persistence engine will still be possible, without changing the client code.
21.2.2.2. Service execution

The `reService` flag of `AllowRemoteExecute: TSQLAllowRemoteExecute` can be used to enable or disable the `Client-Server services via interfaces` (page 419) feature of `mORMot`.

In addition to this global parameter, you can set per-service and per-method `Security` (page 459).

For `Client-Server services via methods` (page 373), if authentication is enabled, any method execution will be processed only from a signed URI.

You can use `TSQLRestServer.ServiceMethodByPassAuthentication()` to disable the need of a signature for a given service method - e.g. it is the case for Auth and Timestamp standard method services.

For `Client-Server services via interfaces` (page 419), if authentication is enabled, any service execution will be processed only from a signed URI.

You can use the `TServiceFactoryServer.ByPassAuthentication` property, to let a given service URI not be signed.

Do not forget to remove authentication for the services for which you want to enable `Scaling via CDN` (page 542). In fact, such world-wide CDN caching services expect the URI to be generic, and not tied to a particular client session.
22. Scripting Engine

22.1. Scripting abilities

As a Delphi framework, mORMot premium language support is for the object pascal language. But it could be convenient to have some part of your software not fixed within the executable. In fact, once the application is compiled, execution flow is written in stone: you can’t change it, unless you modify the Delphi source and compile it again. Since mORMot is Open Source, you can ship the whole source code to your customers or services with no restriction, and diffuse your own code as pre-compiled .dcu files, but your end-user will need to have a Delphi IDE installed (and paid), and know the Delphi language.

This is when scripting does come on the scene. For instance, scripting may allow to customize an application behavior for an end-user (i.e. for reporting), or let a domain expert define evolving appropriate business rules - following Domain-Driven Design (page 99).

If your business model is to publish a core domain expertise (e.g. accounting, peripheral driving, database model, domain objects, communication, AJAX clients...) among several clients, you will sooner or later need to adapt your application to one or several of your customers. There is no "one exe to rule them all". Maintaining several executables could become a "branch-hell". Scripting is welcome here: speed and memory critical functionality (in which mORMot excels) will be hard-coded within the main executable, then everything else could be defined in script.

There are plenty of script languages available. We considered http://code.google.com/p/dwscript, which is well maintained and expressive (it is the
code of our beloved *Smart Mobile Studio*), but is not very commonly used. We still want to include it in the close future.

Then [http://www.lua.org](http://www.lua.org), defines a light and versatile general-purpose language, dedicated to be embedded in any application. Sounds like a viable solution: if you can help with it, your contribution is welcome!

We did also take into consideration [http://www.python.org](http://www.python.org) and [http://www.ruby-lang.org](http://www.ruby-lang.org) but both are now far from light, and are not meant to be embedded, since they are general-purpose languages, with a huge set of full-featured packages.

Then, there is *JavaScript*:

- This is the *World Wide Web* assembler. Every programmer in one way or another knows *JavaScript*;
- *JavaScript* can be a very powerful language - see Crockford’s book *“JavaScript - The Good Parts”*;
- There are a huge number of libraries written in *JavaScript*: template engines (jade, mustache...), SOAP and LDAP clients, and many others (including all node.js libraries of course);
- It was the base for some strongly-typed syntax extensions, like CoffeScript, TypeScript, Dart;
- In case of AJAX / Rich Internet Application we can directly share part of logic between client and server (validation, template rendering...) without any middle-ware;
- One long time mORMot’s user (Pavel, aka mpv) already integrated SpiderMonkey to mORMot’s core. His solution is used on production to serve billion of requests per day, with success. We officially integrated his units. Thanks Pavel!

As a consequence, mORMot introduced direct *JavaScript* support via SpiderMonkey.

It allows to:

- Execute *Delphi* code from *JavaScript* - including our *Object-Relational Mapping (ORM)* (page 92)/ *Service-Oriented Architecture (SOA)* (page 90) methods, or even reporting;
- Consume *JavaScript* code from *Delphi* (e.g. to define and customize any service or rule, or use some existing .js library);
- Expose *JavaScript* objects and functions via a TSMVariant custom variant type: it allows to access any *JavaScript* object properties or call any of its functions via late-binding, from your *Delphi* code, just as if it was written in native Object-Pascal;
- Follow a classic synchronous blocking pattern, rooted on mORMot’s multi-thread efficient model, easy to write and maintain;
- Handle *JavaScript* or *Delphi* objects as Unicode and UTF-8 (page 105) JSON (page 295), ready to be published or consumed via JSON RESTful Client-Server (page 295) remote access.

### 22.2. SpiderMonkey integration

#### 22.2.1. A powerful JavaScript engine

*SpiderMonkey*, the Mozilla *JavaScript* engine, can be embedded in your mORMot application. It could be used on client side, within a *Delphi* application (e.g. for reporting), but the main interest of it may be on the server side.

The word *JavaScript* may bring to mind features such as event handlers (like onclick), DOM objects, window.open, and XMLHttpRequest.

But all of these features are actually not provided by the *SpiderMonkey* engine itself.

*SpiderMonkey* provides a few core *JavaScript* data types—numbers, strings, Arrays, Objects, and so on—and a few methods, such as Array.push. It also makes it easy for each application to expose some of its own objects and functions to *JavaScript* code. Browsers expose DOM objects. Your application will expose objects that are relevant for the kind of scripts you want to write. It is up to the
application developer to decide what objects and methods are exposed to scripts.

### 22.2.2. Direct access to the SpiderMonkey API

The SynSMAPI.pas unit is a tuned conversion of the *SpiderMonkey* API, providing full *ECMAScript* 5 support and *JIT*. The *SpiderMonkey* revision 24 engine is included, with a custom C wrapper around the original C++ code.

You could take a look at [http://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey](http://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey) for a full description of this low-level API, and find our patched version of the library, modified to be published from C instead of C++, in the synsm-mozjs folder of the *mORMot* source code repository.

But the SynSM.pas unit will encapsulate most of it into higher level *Delphi* classes and structures (including a custom variant type), so you probably won't need to use SynSMAPI.pas directly in your code:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSMEngineManager</td>
<td>main access point to the <em>SpiderMonkey</em> per-thread scripting engines</td>
</tr>
<tr>
<td>TSMEngine</td>
<td>implements a Thread-Safe <em>JavaScript</em> engine instance</td>
</tr>
<tr>
<td>TSMObject</td>
<td>wrap a <em>JavaScript</em> object and its execution context</td>
</tr>
<tr>
<td>TSMValue</td>
<td>wrap a <em>JavaScript</em> value, and interfaces it with <em>Delphi</em> types</td>
</tr>
<tr>
<td>TSMVariant, TSMVariantData</td>
<td>define a custom variant type, for direct access to any <em>JavaScript</em> object, with late-binding</td>
</tr>
</tbody>
</table>

We will see how to work with all those classes.

### 22.2.3. Execution scheme

The *SpiderMonkey* *JavaScript* engine compiles and executes scripts containing *JavaScript* statements and functions. The engine handles memory allocation for the objects needed to execute scripts, and it cleans up—garbage collects—objects it no longer needs.

In order to run any *JavaScript* code in *SpiderMonkey*, an application must have three key elements:
- A JSRuntime,
- A JSContext,
- And a global JSObject.

A JSRuntime, or runtime, is the space in which the *JavaScript* variables, objects, scripts, and contexts used by your application are allocated. Every JSContext and every object in an application lives within a JSRuntime. They cannot travel to other runtimes or be shared across runtimes.

A JSContext, or context, is like a little machine that can do many things involving *JavaScript* code and objects. It can compile and execute scripts, get and set object properties, call *JavaScript* functions, convert *JavaScript* data from one type to another, create objects, and so on.

Lastly, the global JSObject is a *JavaScript* object which contains all the classes, functions, and variables that are available for *JavaScript* code to use. Whenever a web browser code does something like `window.open("http://www.mozilla.org/")`, it is accessing a global property, in this case `window`. *SpiderMonkey* applications have full control over what global properties scripts can see.

Every *SpiderMonkey* instance starts out every execution context by creating its JSRuntime, JSContext
instances, and a global JSObject. It populates this global object with the standard JavaScript classes, like Array and Object. Then application initialization code will add whatever custom classes, functions, and variables (like window) the application wants to provide; it may be, for a mORMot server application, ORM access or SOA services consumption and/or implementation.

Each time the application runs a JavaScript script (using, for example, JS_EvaluateScript), it provides the global object for that script to use. As the script runs, it can create global functions and variables of its own. All of these functions, classes, and variables are stored as properties of the global object.

### 22.2.4. Creating your execution context

The main point about those three key elements is that, in the current implementation pattern of SpiderMonkey, runtime, context or global objects are not thread-safe.

Therefore, in the mORMot's use of this library, each thread will have its own instance of each.

In the SynSM.pas unit, a TSMEngine class has been defined to give access to all those linked elements:

```pascal
TSMEngine = class
  ...  
  /// access to the associated global object as a TSMVariant custom variant
  // - allows direct property and method executions in Delphi code, via
  // Late-binding
  property Global: variant read FGlobal;
  /// access to the associated global object as a TSMObject wrapper
  // - you can use it to register a method
  property GlobalObject: TSMObject read FGlobalObject;
  /// access to the associated global object as low-level PJSObject
  property GlobalObj: PJSObject read FGlobalObject.fobj;
  /// access to the associated execution context
  property cx: PJSContext read fCx;
  /// access to the associated execution runtime
  property rt: PJSRuntime read frt;
  ...
```

Our implementation will define one Runtime, one Context, and one global object per thread, i.e. one TSMEngine class instance per thread.

A JSRuntime, or runtime, is created for each TSMEngine instance. In practice, you won't need access to this value, but rely either on a JSContext or directly a TSMEngine.

A JSContext, or context, will be the main entry point of all SpiderMonkey API, which expect this context to be supplied as parameter. In mORMot, you can retrieve the running TSMEngine from its context by using the function TSMObject.Engine: TSMEngine - in fact, the engine instance is stored in the private data slot of each JSContext.

Lastly, the TSMEngine's global object contains all the classes, functions, and variables that are available for JavaScript code to use. For a mORMot server application, ORM access or SOA services consumption and/or implementation, as stated above.

You can note that there are several ways to access this global object instance, from high-level to low-level JavaScript object types. The TSMEngine.Global property above is in fact a variant. Our SynSM.pas unit defines in fact a custom variant type, identified as the TSMVariant class, able to access any JavaScript object via late-binding, for both variables and functions:

```pascal
engine.Global.MyFunction(1,'text');
```

Most web applications only need one runtime, since they are running in a single thread - and (ab)use
of callbacks for non-blocking execution. But in mORMot, you will have one TMS Engine instance per thread, using the TMS Engine Manager.ThreadSafeEngine method. Then all execution may be blocking, without any noticeable performance issue, since the whole mORMot threading design was defined to maximize execution resources.

22.2.5. Blocking threading model

This threading model is the big difference with other server-side scripting implementation schemes, e.g. the well-known node.js solution.

Multi-threading is not evil, when properly used. And thanks to the mORMot's design, you won't be afraid of writing blocking JavaScript code, without any callbacks. In practice, those callbacks are what makes most JavaScript code difficult to maintain.

On the client side, i.e. in a web browser, the JavaScript engine only uses one thread per web page, then uses callbacks to defer execution of long-running methods (like a remote HTTP request). If fact, this is one well identified performance issue of modern AJAX applications. For instance, it is not possible to perform some intensive calculation in JavaScript, without breaking the web application responsiveness: you have to split your computation task in small tasks, then let the JavaScript code pause, until a next piece of computation could be triggered... On server side, node.js allows to define Fibers and Futures - see http://github.com/laverdet/node-fibers.. - but this is not available on web clients. Some browsers did only start to uncouple the JavaScript execution thread from the HTML rendering thread - and even this is hard to implement... we reached here the limit of a technology rooted in the 80's...

On the server side, node.js did follow this pattern, which did make sense (it allows to share code with the client side, with some name-space tricks), but it is also a big waste of resources. Why should we stick to an implementation pattern inherited from the 80's computing model, when all CPUs were mono core, and threads were not available?

The main problem when working with one single thread, is that your code shall be asynchronous. Soon or later, you will face a syndrome known as "Callback Hell". In short, you are nesting anonymous functions, and define callbacks. The main issue, in addition to lower readability and being potentially sunk into function() nesting, is that you just lost the JavaScript exception model. In fact, each callback function has to explicitly check for the error (returned as a parameter in the callback function), and handle it.

Of course, you can use so-called Promises and some nice libraries - mainly async.js.

But even those libraries add complexity, and make code more difficult to write. For instance, consider the following non-blocking/asynchronous code:

```javascript
getTweetsFor("domenic") // promise-returning function
  .then(function (tweets) {
    var shortUrls = parseTweetsForUrls(tweets);
    var mostRecentShortUrl = shortUrls[0];
    return expandUrlUsingTwitterApi(mostRecentShortUrl); // promise-returning function
  })
  .then(httpGet) // promise-returning function
  .then(
    function (responseData) {
      console.log("Most recent link text:", responseData);
    },
    function (error) {
      console.error("Error with the twitterverse:", error);
    }
  );
```
This kind of code will be perfectly readable for a *JavaScript* daily user, or someone fluent with functional languages.

But the following blocking/synchronous code may sound much more familiar, safer and less verbose, to most *Delphi* / *Java* / *C#* programmer:

```javascript
try {
    var tweets = getTweetsFor("domenic"); // blocking
    var shortUrls = parseTweetsForUrls(tweets);
    var mostRecentShortUrl = shortUrls[0];
    var responseBody = httpGet(expandUrlUsingTwitterApi(mostRecentShortUrl)); // blocking x 2
    console.log("Most recent link text:", responseBody);
    catch (error) {
        console.error("Error with the twitterverse: ", error);
    }
}
```

Thanks to the blocking pattern, it becomes obvious that code readability and maintainability is as high as possible, and error detection is handled nicely via *JavaScript* exceptions, and a global try .. catch.

Last but not least, debugging blocking code is easy and straightforward, since the execution will be linear, following the code flow.

Upcoming ECMAScript 6 should go even further thanks to the `yield` keyword and some task generators - see [http://taskjs.org](http://taskjs.org) - so that asynchronous code may become closer to the synchronous pattern. But even with `yield`, your code won't be as clean as with plain blocking style.

In *mORMot*, we did choose to follow an alternate path, i.e. write blocking synchronous code. Sample above shows how easier it is to work with. If you use it to define some huge business logic, or let a domain expert write the code, blocking syntax is much more straightforward.

Of course, *mORMot* allows you to use callbacks and functional programming pattern in your *JavaScript* code, if needed. But by default, you are allowed to write KISS blocking code.

### 22.3. Interaction with existing code

Within *mORMot* units, you can mix *Delphi* and *JavaScript* code by two ways:

- Either define your own functions in *Delphi* code, and execute them from *JavaScript*;
- Or define your own functions in *JavaScript* code (including any third-party library), and execute them from *Delphi*.

Like for other part of our framework, performance and integration has been tuned, to follow our KISS way.

You can take a look at "22 - JavaScript HTTPApi web server\JSHttpApiServer.dpr" sample for reference code.

#### 22.3.1. Proper engine initialization

As was previously stated, the main point to interface the *JavaScript* engine is to register all methods when the TSMEngine instance is initialized.

For this, you set the corresponding OnNewEngine callback event to the main TSMEngineManager instance.

See for instance, in the sample code:

```delphi
constructor TTestServer.Create(const Path: TFileName);
```
In DoOnNewEngine, you will initialize every newly created TSMEngine instance, to register all needed Delphi methods and prepare access to JavaScript via the runtime's global JSObject.

Then each time you want to access the JavaScript engine, you will write for instance:

```delphi
function TTestServer.Process(Ctxt: THttpServerRequest): cardinal;
var
  engine: TSMEngine;
...
  engine := fSMManager.ThreadSafeEngine;
  // now you can use engine, e.g. engine.Global.someMethod()
end;
```

Each thread of the HTTP server thread-pool will be initialized on the fly if needed, or the previously initialized instance will be quickly returned otherwise.

Once you have the TSMEngine instance corresponding to the current thread, you can launch actions on its global object, or tune its execution.

For instance, it could be a good idea to check for the JavaScript VM's garbage collector:

```delphi
function TTestServer.Process(Ctxt: THttpServerRequest): cardinal;
...
  engine := fSMManager.ThreadSafeEngine;
  engine.MaybeGarbageCollect; // perform garbage collection if needed
end;
```

We will now find out how to interact between JavaScript and Delphi code.

### 22.3.2. Calling Delphi code from JavaScript

In order to call some Delphi method from JavaScript, you will have to register the method. As just stated, it is done by setting a callback within TSMEngineManager.OnNewEngine initialization code. For instance:

```delphi
procedure TTestServer.DoOnNewEngine(const Engine: TSMEngine);
...
  // add native function to the engine
  Engine.RegisterMethod(Engine.GlobalObj, 'loadFile', LoadFile, 1);
end;
```

Here, the local LoadFile() method is implemented as such in native code:

```delphi
function TTestServer.LoadFile(const This: variant; const Args: array of variant): variant;
begin
  if length(Args)<>1 then
    raise Exception.Create('Invalid number of args for loadFile(): required 1 (file path)');
  result := AnyTextFileToSynUnicode(Args[0]);
end;
```

As you can see, this is perfectly easy to follow. Its purpose is to load a file content from JavaScript, by defining a new global function named loadFile().

Remember that the SpiderMonkey engine, by itself, does not know anything about file system, database or even DOM. Only basic objects were registered, like arrays. We have to explicitly register the functions needed by the JavaScript code.

In the above code snippet, we used the TSMEngineMethodEventVariant callback signature, marshalling variant values as parameters. This is the easiest method, with only a slight performance impact.
Such methods have the following features:

- Arguments will be transmitted from JavaScript values as simple Delphi types (for numbers or text), or as our custom TSMVariant type for JavaScript objects, which allows late-binding;
- The This: variant first parameter map the "callee" JavaScript object as a TSMVariant custom instance, so that you will be able to access the other object's methods or properties directly via late-binding;
- You can benefit of the JavaScript feature of variable number of arguments when calling a function, since the input arguments is a dynamic array of variant;
- All those registered methods are registered in a list maintained in the TSMEngine instance, so it could be pretty convenient to work with, in some cases;
- You can still access to the low-level JSObject values of any the argument, if needed, since they can be trans-typed to a TSMVariantData instance (see below) - so you do not loose any information;
- The Delphi native method will be protected by the mORMot wrapper, so that any exception raised within the process will be catch and transmitted as a JavaScript exception to the runtime;
- There is also an hidden set of the FPU exception mask during execution of native code (more on it later on) - you should not bother on it here.

Now consider how you should have written the same loadFile() function via low-level API calls.

First, we register the callback:

```delphi
procedure TTestServer.DoOnNewEngine(const Engine: TSMEngine);
...
// add native function to the engine
Engine.GlobalObject.DefineNativeMethod('loadFile', nsm_loadFile, 1);
end;
```

Then its implementation:

```delphi
function nsm_loadFile(cx: PJSContext; argc: uintN; vp: Pjsval): JSBool;
cdecl;
var
  in_argv: PjsvalVector;
  filePath: TFileName;
begin
  TSynFPUException.ForDelphiCode;
  try
    if argc<>1 then
      raise Exception.Create('Invalid number of args for loadFile(): required 1 (file path)');
    in_argv := JS_ARGV(cx,vp);
    filePath := JSVAL_TO_STRING(in_argv[0]).ToString(cx);
    JS_SET_RVAL(cx, vp, cx^.NewJSString(AnyTextFileToSynUnicode(filePath)).ToJSVal);
    Result := JS_TRUE;
  except
    on E: Exception do begin
      // all exceptions MUST be catched on Delphi side
      JS_SET_RVAL(cx, vp, JSVAL_VOID);
      JSError(cx, E);
      Result := JS_FALSE;
    end;
  end;
end;
```

As you can see, this nsm_loadFile() function is much more difficult to follow:

- Your code shall begin with a cryptic TSynFPUException.ForDelphiCode instruction, to protect the FPU exception flag during execution of native code (Delphi RTL expects its own set of FPU exception mask during execution, which does not match the FPU exception mask expected by SpiderMonkey);
- You have to explicitly catch any Delphi exception which may raise, with a try...finally block, and marshal them back as JavaScript errors;
- You need to do a lot of manual low-level conversions - via JS_ARGV() then e.g. JSVAL_TO_STRING() macros - to retrieve the actual values of the arguments;
- And the returning function is to be marshaled by hand - see the JS_SET_RVAL() line.
Since the variant-based callback has only a slight performance impact (nothing measurable, when compared to the SpiderMonkey engine performance itself), and still have access to all the transmitted information, we strongly encourage you to use this safer and cleaner pattern, and do not define any native function via low-level API.

Note that there is an alternate JSON-based callback, which is not to be used in your end-user code, but will be used when marshalling to JSON is needed, e.g. when working with mORMot's ORM or SOA features.

### 22.3.3. TSMVariant custom type

As stated above, the SynSM.pas unit defines a TSMVariant custom variant type. It will be used by the unit to marshal any JSObject instance as variant.

Via the magic of late-binding, it will allow access of any JavaScript object property, or execute any of its functions. Only with a slightly performance penalty, but with much better code readability than with low-level access of the SpiderMonkey API.

The TSMVariantData memory structure can be used to map such a TSMVariant variant instance. In fact, the custom variant type will store not only the JSObject value, but also its execution context - i.e. JSContext - so is pretty convenient to work with.

For instance, you may be able to write code as such:

```delphi
defunction TMyClass.MyFunction(const This: variant; const Args: array of variant): variant;
var
global: variant;
begin
  TSMVariantData(This).GetGlobal(global);
  // same as:
  global := TSMVariantData(This).SMObject.Engine.Global;
  // but you may also write directly:
  with TSMVariantData(This).SMObject.Engine do
  begin
    global.anotherFunction(Args[0],Args[1],"test");
    result := AnyTextFileToSynUnicode(Args[0]);
  end;
end;
```

Here, the This custom variant instance is trans-typed via TSMVariantData(This) to access its internal properties.

### 22.3.4. Calling JavaScript code from Delphi

In order to execute some JavaScript code from Delphi, you should first define the JavaScript functions to be executed.

This shall take place within TSEngineManager.OnNewEngine initialization code:

```delphi
procedure TTestServer.DoOnNewEngine(const Engine: TSMEngine);
var
  showDownRunner: SynUnicode;
begin
  // add external JavaScript library to engine (port of the Markdown library)
  Engine.Evaluate(fShowDownLib, 'showdown.js');
  // add the bootstrap function calling loadfile() then showdown’s makeHtml()
  showDownRunner := AnyTextFileToSynUnicode(ExeVersion.ProgramFilePath+'showDownRunner.js');
  Engine.Evaluate(showDownRunner, 'showDownRunner.js');
  ...
end;
```

This code first evaluates (i.e. "executes") a general-purpose JavaScript library contained in the showdown.js file, available in the sample executable folder. This is an open source library able to convert any Markdown markup into HTML. Plain standard JavaScript code.
Then we evaluate (i.e. "execute") a small piece of JavaScript code, to link the makeHtml() function of the just defined library with our loadFile() native function:

```javascript
function showDownRunner(pathToFile){
    var src = loadFile(pathToFile); // call Delphi native code
    var converter = new Showdown.converter(); // get the Showdown converted
    return converter.makeHtml(src); // convert .md content into HTML via showdown.js
}
```

Now we have a new global function `showDownRunner(pathToFile)` at hand, ready to be executed by our Delphi code:

```delphi
function TTestServer.Process(Ctxt: THttpServerRequest): cardinal;
var
    content: variant;
    FileName, FileExt: TFileName;
    engine: TSMEngine;
    ...
begin
    if FileExt='.md' then begin
        engine := fSMManager.ThreadSafeEngine;
        content := engine.Global.showDownRunner(FileName);
        ...
end;
```

As you can see, we access the function via late-binding. Above code is perfectly readable, and we call here a JavaScript function and a whole library as natural as if it was native code.

Without late-binding, we may have written, accessing not the Global TSMVariant instance, but the lower level GlobalObject: TSMObject property:

```delphi
content := engine.GlobalObject.Run('showDownRunner',[SynUnicode(FileName)]);
```

It is up to you to choose which kind of code you prefer, but late-binding is worth considering.
As we have seen when dealing about Security (page 544), the framework offers built-in encryption of the content transmitted between its REST client and server sides, especially via Custom Encodings (page 333), or HTTPS. The later, when using TLS 1.2 and proven patterns, implements state-of-the-art security. But default mORMot encryption, even if using proven algorithms like AES256-CFB and SHA256, uses symmetric keys, that is the same secret key is shared on both client and server sides.

Asymmetric encryption, also known as public-key cryptography, uses pairs of keys:
- Public keys that may be disseminated widely;
- Paired with private keys which are known only to the owner.

The framework features a full asymmetric encryption system, based on Elliptic curve cryptography (ECC), which may be used at application level (i.e. to protect your application data), or at transmission level (to enhance communication safety).
23.1. Public-key Cryptography

Once you have generated a public/private pairs of keys, you can perform two functions:
- **Authenticate** a message originated with a holder of the private key; a certification system should be used to maintain a trust chain of authority;
- **Encrypt** a message with a public key to ensure that only the holder of the paired private key can decrypt it.

23.1.1. Keys Generation and Distribution

First process is to generate a pair of public/private keys. Some random number generator, probably based on an external entropy source, will gather unpredictable numbers, which will be consumed by a public-key algorithm to generate the actual set of keys. This step usually requires some computing powers, due to the complexity of the algorithms involved, and the encryption needed for storing the private key in secret.

Let's explain how it works for the classic Alice/Bob scheme:

![Asymmetric Key Generation Diagram]

Now we have two pairs of keys:
- `alice.public` and `alice.private` for Alice;
- `bob.public` and `bob.private` for Bob.

By design, public keys (`alice.public` and `bob.public`) can be published, via mail, in application settings, as unprotected file, or even on a public server. On the contrary, private keys (`alice.private` or `bob.private`) should remain as secret as possible, and are usually encrypted, then stored in password-protected files, in some safe place of the operating system, or even dedicated hardware.

In practice, Alice will send her `alice.public` key to Bob, so that:
- Bob can verify the digital signature of a message sent by Alice, who signed it with her `alice.private` key;
- Bob can encrypt some information with the known `alice.public` key, then send it to Alice - and that only Alice could decrypt it using her `alice.private` key.

Of course, since Bob has his own set of keys, he also publishes his `bob.public` key, so that:
- Alice can verify the digital signature of a message sent by Blob, who signed it with his `bob.private` key;
- Alice can encrypt some information with the known `bob.public` key, then send it to Bob - and that only Bob could decrypt it using his `bob.private` key.

Key distribution is an important part of any asymmetric encryption scheme. The whole security chain...
is as secure as its weakest link, so the secrecy of the private keys requires as most attention as possible. Every software solution using security will probably require external audits, at least peer review, to validate each implementation.

23.1.2. Message Authentication

Any kind of message (most probably a file or a memory buffer) can be authenticated using digital signatures, using the private key of the sender. Then, on the other side, the receiver can verify the message signature, using the public key of the sender.

Asymmetric Digital Signature

As you can see, if Bob believes that the alice.public file comes from Alice, he can assume that the messate.txt content has really been sent by Alice. Most of the time, in such simple scenarios, Alice probably gave directly her alice.public file to Bob, for instance via an email. But for most complex scenarios, like the Client/Server solutions which can be built using the mORMot framework, the multiplicity of nodes, and therefore keys, induces a potential risk.

23.1.3. Certificates and Public Key Infrastructure

A central problem with the use of public-key cryptography is confidence that a particular public key is authentic, in that it belongs to the person or entity claimed, and has not been tampered with or replaced by a malicious third party. Digital signature is more than just creating a hash of some content, or applying some kind of "seal" on it: validation should be done against some reference public keys, which are hosted into a public-key infrastructure (PKI). One or more third-parties, aka certification authorities (CA), certify ownership of key pairs, by supplying some online service and/or local safe storage of reference public keys, but keeping their own private keys secret. Any certificate authority could sign a message with his private key, or even delegate his own authority to another certification instance, by signing the intermediate authority with his private key. If one certificate is compromised - i.e. if its private key has been released - the whole chain of trust is broken, and all dependent certificates should be immediately revoked.

In practice, when a public certificate key is generated in such a trusted PKI system, it will contain:
- The genuine public key material, depending on the underlying algorithm used;
- Some ownership information (i.e. who emitted it);
- The scope of the certification (may apply to a user, a company, a web site, an application...);
- A certified link to one or several other certificates, signed with their private key to prove their authenticity using the known public key of the CA chain;
- Optional validity and revocation dates - since it is a good practice to renew certificates on a regular
The private key store may also contain the very same set of information, added to its private key material. It will enforce consistency between public and private keys - for instance, you won't be fooled by using a private key after its associated public certificates expired.

Certification authorities create a chain of trust, used as reference to authenticate public keys. Every Operating System, or Internet browser do contain some root certificates, and the whole Internet security (HTTPS/TLS) is governed by such a PKI. Of course, for your own set of applications or products, you can create your own key chain, keeping the same principles - mainly private key secrecy and trust chain management.

23.1.4. Message encryption

A naive approach of hiding a message content is by using a secret key or pattern, then apply it on the message. It has been done since ages, and will be safe as much as the symmetric key is safe. As a side effect, you have to trust the receiver not to spread the key to the public - and in fact, you shouldn’t: don’t trust anyone, even not you!

Public-key cryptography solves this problem by using a public key to encrypt a message, which will therefore only be decryptable by someone knowing the corresponding secret key.

![Asymmetric Encryption Scheme](image)

Of course, you can not only encrypt the message, but also sign it, using the other end public key. Here is how a sign-then-encrypt pattern can be implemented:
Asymmetric Sign-Then-Encrypt Scheme

As always, the alice.public and bob.public keys are validated against the trust chain of certificates of a public-key infrastructure (PKI).

With all this elements, we can now apply them to our mORMot applications.
23.2. Elliptic Curve Cryptography

The framework features an implementation of *Elliptic Curve Cryptography* (ECC), based on the mathematical structure of the "elliptic curve discrete logarithm problem". The mathematical community has not made any major progress in improving algorithms to solve this problem since it was independently introduced by Koblitz and Miller in 1985. In short, the public key is an equation for an elliptic curve and a point that lies on that curve. The private key is a number. Thanks to the symmetry of the elliptic curve, there is some kind of symmetry also between ECC public and private key values, and ECDSA and ECDH algorithms capitalize on this characteristic to compute a digital signature or a shared secret.

In comparison to the RSA algorithm, ECC has some advantages:
- Smaller key size, for the same level of safety (a 256-bit elliptic curve key is comparable to a 3072-bit RSA key);
- Well endorsed by most certification authorities (NIST/NSA);
- Faster performance, especially when the key size increases;
- Offers perfect forward secrecy, since a fresh key is created for every encryption;
- Potentially less patents infringement, in all its practical appliances;
- Last but not least, it is one the strongest algorithms for the future of web.

There will no doubt be criticism of our decision to re-implement a whole public-key cryptography stack from scratch, with its own small choice of algorithms, instead of using an existing library (like OpenSSL), and established standards (like X509).

To be fair, such libraries are complex and confusing, whereas we selected a set of future-proof algorithms (AES256 excluding ECB, HMAC-SHA256, PBKDF2_HMAC_SHA256, ECDSA, ECIES...) to follow *mORMot*'s KISS and DRY principles, keep code maintainable and readable, and reduce risk assessment scope. We followed all identified best practices, and tried to avoid, from the beginning, buffer overflows, weak protocols, low entropy, low default values, serial collision, forensic vulnerabilities, hidden memory copies, evil optimizations. The last thing we want to do is to start mandating DLLs, which are perhaps deprecated/unsafe if part of the OS. Last but not least, it was fun, we learned a lot, and we hope you will enjoy using it, and contribute to it!

23.2.1. Introducing SynEcc

The *mORMot*'s SynEcc.pas unit implements full ECC computation, using secp256r1 curve, i.e. NIST P-256, or OpenSSL's prime256v1. The low-level computation is done in optimized C code - from the [https://github.com/esxgx/easy-ecc](https://github.com/esxgx/easy-ecc) project - and is statically linked in your Windows or Linux executable: i.e. no external .dll/.so library is needed. On targets (e.g. BSD/MacOSX or ARM) where we didn't provide the static .o files, there is an optimized pascal version available. Then we defined a feature-rich set of object pascal classes on top of this solid ECC ground, to include certificates, safe storage of private keys, JSON publication of public keys, as an integrated toolset.

All needed low-level asymmetric cryptography is available:
- ECC key generation, using SynCrypto.pas's secure TAESPRNG as random seed;
- ECDSA signature and verification of 256-bit hashes;
- ECDH shared secret computation - suitable for ECIES encryption, after PBKDF2_HMAC_SHA256 derivation.

The very same SynEcc.pas unit defines some high-level classes and structures, ready to implement:
- Authority certificates - via public TECCCertificate and private TECCCertificateSecret classes, and full PKI chaining - see TECCCertificateChain;
- Digital signature of files or memory buffers - via TECCSignatureCertified;
- Encryption of files or memory buffers - calling TECCCertificate.Encrypt and TECCCertificateSecret.Decrypt methods;
- Innovative .cheat files generation, for safe storage of private keys passwords, encrypted from a master cheat .public key and its master password.

You are free to use those classes, in your programs, whenever some advanced cryptography is needed - and it will eventually be the case, trust me! A command-line ECC tool has also been developed, for convenient operation on files.

### 23.2.2. ECC command line tool

You will find in the SQLite3\Samples\33 - ECC folder the source code of the ECC.dpr console project. Just compile it into an executable, accessible from your command line prompt. Or download an already compiled version from https://synopse.info/files/ecc.7z.

It works with no problem under Windows, or Linux, with no external dependency (e.g. no OpenSSL needed), so could be used in an automated server infrastructure. No need to deploy a complex PKI system, just manage your certificates, encryption and signature details, via a single command line tool.

If you run it without argument, you will get simple help information (here is the list at the time of this writing, your own version may differ):

```bash
>ecc

Synopse ECC certificate-based public-key cryptography

---------------------------------
Using mORMot's SynECC rev. 1.18.3112

ECC help
ECC new -auth key.private -authpass authP@ssW0rd -authrounds 60000
  -issuer toto@toto.com -start 2016-10-30 -days 30
  -newpass P@ssW0rd@ -newrounds 60000
ECC rekey -auth key.private -authpass P@ssW0rd -authrounds 60000
  -newpass newP@ssW0rd@ -newrounds 60000
ECC sign -file some.doc -auth key.private -pass P@ssW0rd -rounds 60000
ECC verify -file some.doc -auth key.public
ECC source -auth key.private -pass P@ssW0rd -rounds 60000
  -const MY_PRIVKEY -comment "My Private Key"
ECC infopriv -auth key.private -pass P@ssW0rd -rounds 60000
  -auth round -key -public
ECC chainfile1.public file2.public file3.public ...
ECC chainall
ECC crypt -file some.doc -out some.doc.synec -auth key.public
  -salt Pass salt -saltrounds 60000
ECC decrypt -file some.doc.synec -out some.doc -auth key.private
  -authpass P@ssW0rd -authrounds 60000 -salt Pass salt -saltrounds 60000
ECC infocrypt -file some.doc.synec
ECC cheatinit -newpass MasterP@ssW0rd@ -newrounds 100000
ECC cheat -auth key.private -authpass MasterP@ssW0rd@ -authrounds 100000

Note that you can add the -noprompt switch for no console interactivity.
```

As you can see, the action is defined by a keyword, at first place (new sign verify source...). Then some optional parameters, in form of -key value pairs, can be supplied. If no parameter is specified, the ECC console application will prompt for input, with user-friendly questions, and adequate default values.

You can define the -noprompt switch to force no console interaction at all, therefore allowing
automated use from another process, or batch file. The ECCProcess.pas unit publishes all high-level commands of the ECC tool, so could be reused in your own setup or maintenance projects.

We will now use this ECC tool to show most common features of the SynEcc unit, but also showing the code corresponding to each action.

23.2.3. Keys and Certificates Generation

The first step is to create a new key pair, which will contain their own certification information:

```
> ecc new
Enter the first chars of the .private file name of the signing authority.
Will create a self-signed certificate if left void.
Auth:

Enter Issuer identifier text.
Will be truncated to 15-20 ascii-7 chars.
Issuer [arbou] :

Enter the YYYY-MM-DD start date of its validity.
@ will create a never-expiring certificate
Start [2016-09-23] :

Enter the number of days of its validity.
Days [365] :

Enter a private PassPhrase for the new key (at least 8 chars long).
Save this in a safe place: if you forget it, the key will be useless!
NewPass [#weLHn5E.Qfe] :

Enter the PassPhrase iteration round for the new key (at least 1000).
The higher, the safer, but will demand more computation time.
NewRounds [60000] :

Corresponding TSynPersistentWithPassword.ComputePassword:
encryption ErMdwaro/8JfsC2ZC
authMutual SqMx5M1v+O71+VY95zkhU2wP791KL3s1BFnd+a
authServer SqMhx6Glv+O71+VY95zkhU2wP791KL3s1BFnd+a
authClient SqMlx6Glv+O71+VY95zkhU2wP791KL3s1BFnd+a

8BC90201EF55EE34F62DBAD8FE8CF14DC.public/.private file created.

Here we keep the default values, including the safe generated password (#weLHn5E.Qfe). You should write down this password in a safe place, because it will be required for any use of the private key, e.g. when signing or decrypting a message. If you forget about this password, there will be no way of accessing this private key any more - you have been warned!

We will see below (page 587) how enabling the ECC cheat mode may help storing the generated .private key passwords in a .cheat encrypted local file using a cheat.public key, to safely recover a password, from a master cheat.private key and its associated password.

The last line contains the identifier (or serial) of the generated key. This hexadecimal value (8BC90201EF55EE34F62DBAD8FE8CF14DC) will be used externally to identify the key, and internally (within other certificates) to map this particular key. Note that you do not need to type all the characters of the serial in the ECC tool: only the first characters are enough (e.g. 8BC9), as soon as they identify one unique file in the current folder.

You can check the generated files in the current folder:

```
> dir *.p*
23/09/2016  13:46  2 320 8BC90201EF55EE34F62DBAD8FE8CF14DC.private
```
The .private file is some raw binary content, encrypted using the #weLHn5E.Qfe password. The .public file, on the contrary, is stored as a plain JSON object:

```json
{
    "Version": 1,
    "Serial": "8BC90201EF55EE34F62DBA8FE8CF14DC",
    "Issuer": "arbou",
    "IssueDate": "2016-09-23",
    "ValidityStart": "2016-09-23",
    "ValidityEnd": "2017-09-23",
    "AuthoritySerial": "8BC90201EF55EE34F62DBA8FE8CF14DC",
    "AuthorityIssuer": "arbou",
    "IsSelfSigned": true,
    "Base64": "AQA1ADUAogGLyQIB71XuNPYtuo/ozxTcGr0DgaAAAAAAAAAAAAAAAIVJAgHvVe409i26j+jPFNwas40AAA
AAAAAAAAAAAAAAAQjxOQnzw2SjFLOw8qvWvyyvme8ncIVS1/MmNg+I24wKnxAi6oB87hJ/tpo3zTjUQDssB8AIId+/QR
SF15RccuOy/j/ebexb6xANtZEO3dT/swBUjQy/CYIVQeSTSDZYpQAAAA"
}
```

You can see all information stored in a TECCCertificate instance. The "Base64" field is in fact a raw serialization of the whole content, so its string value contains all information of a public certificate, e.g. in application settings.

We did not specify any authority at the first Auth: prompt. As a result, this key pair will be a self-signed certificate - see the "IsSelfSigned": true field in the above JSON, and that "Serial" and "AuthoritySerial" identifiers do match. We will use it as root certificate to create a certificate chain.

All further certificates will eventually be signed by this root authority. For instance:

```bash
> ecc new
Enter the first chars of the .private file name of the signing authority. Will create a self-signed certificate if left void.
Auth: 8

Will use: 8BC90201EF55EE34F62DBA8FE8CF14DC.private

Enter the PassPhrase of this .private file.
AuthPass: #weLHn5E.Qfe

Enter the PassPhrase iteration rounds of this .private file.
AuthRounds [60000]:

Enter Issuer identifier text. Will be truncated to 15-20 ascii-7 chars.
Issuer [arbou]: toto

Enter the YYYY-MM-DD start date of its validity. 0 will create a never-expiring certificate.
Start [2016-09-23]: 0

Enter a private PassPhrase for the new key (at least 8 chars long). Save this in a safe place: if you forget it, the key will be useless!
NewPass [b3dEB+Dw8B3d]:

Corresponding TSynPersistentWithPassword.ComputePassword:
cIK5hkJDu5/98mm

Enter the PassPhrase iteration round for the new key (at least 1000). The higher, the safer, but will demand more computation time.
NewRounds [60000]:
```
As you can see, we entered just 8 for the first Auth: prompt, and the tool identified the single 8*.private file in the current folder. Then we entered its associated #weLhn5E.Qfe password - any wrong password would have broken the generation. This authority will never expire by itself (we entered 0 as Start: prompt) - but since its root certificate has an expiration date, it will expire when the root expires.

Now we can see the two sets of keys:

```plaintext
> dir *.p*
23/09/2016  14:27             2 320 03B8865C6B982A39E9EFB1DC1A95D227.private
23/09/2016  14:27               524 03B8865C6B982A39E9EFB1DC1A95D227.public
23/09/2016  13:46             2 320 8BC90201EF55EE34F62DBA8FE8CF14DC.private
23/09/2016  13:46               544 8BC90201EF55EE34F62DBA8FE8CF14DC.public
```

The newly created .public file contains:

```json
{
  "Version": 1,
  "Serial": "03B8865C6B982A39E9EFB1DC1A95D227",
  "Issuer": "toto",
  "IssueDate": "2016-09-23",
  "ValidityStart": "",
  "ValidityEnd": "",
  "AuthoritySerial": "8BC90201EF55EE34F62DBA8FE8CF14DC",
  "AuthorityIssuer": "arbou",
  "IsSelfSigned": false,
  "Base64": "AQA1AAAAAAADuIZca5gqOenvsvsdwaidInhiGAAAAAAAIAiV3AgHvVe409i26j+jjPFNwas4OAAA AAAAAAAAAsAr2rPT7xPmChxIt3+71BFYVAPBPwLcsR6uWoyndrQfI7S57iY7q1RfKgpe6EmEENX4Ww07Exu309S5 OAG/EHau0BtwqzBNHJeigQgKadb6ioMwasbKtj3BNMFx7EMIsnxnA4AAAA"
}
```

You can recognize the expected values of "Serial", "AuthoritySerial" and "IsSelfSigned" fields.

We could create a certificates chain of all available keys in the current folder, by running:

```plaintext
> ecc chainall
chain.ca file created.
```

The chain.ca file is a JSON object, containing all public information of the whole certificates chain, with the "PublicKey64" JSON array ready to be copied and pasted in your applications settings or source, then used via the TECcertificateChain class:

```json
{
  "PublicKeyBase64": [
    "AQA1AAAAAA...",
    "AQA1ADUAogG..."
  ],
  "Items": [
    {
      "Version": 1,
      "Serial": "03B8865C6B982A39E9EFB1DC1A95D227",
      "Issuer": "toto",
      "IssueDate": "2016-09-23",
      "ValidityStart": "",
      "ValidityEnd": "",
      "AuthoritySerial": "8BC90201EF55EE34F62DBA8FE8CF14DC",
      "AuthorityIssuer": "arbou",
      "IsSelfSigned": false
    },
    {
      "Version": 1,
      "Serial": "8BC90201EF55EE34F62DBA8FE8CF14DC",
    }
  ]
}
```
In the above sample, we cut down the "PublicBase64" values, to save some paper and trees. They map the content already shown in the .public JSON files. In fact, the same information is stored two times: once in "PublicBase64", and another time in each individual properties ("Version", "Serial", "Issuer"...) of the "Items" items.

An easy way of keys management is to keep a safe mean of storage (e.g. a pair of USB pen-drives, with at least one kept in a physical vault), then put all your certificate chains in dedicated folders. All public keys - i.e. *.public and chain.ca files - are meant to be public, so could be spread away everywhere. Just keep an eye on your .private files, and their associated passwords. A hardware-secured drive may be an overkill, since the .private files are already encrypted and password-protected with state-of-the-art software protection, i.e. AFsplit anti-forensic diffusion and AES256-CFB encryption on a PBKDF2_HMAC_SHA256 derived password, with a huge number of rounds (60000).

Remember that often, the weakest link of the security chain is between the chair and the keyboard, not within the computer. Do not reuse passwords between keys, and remember you have a "rekey" command available on the ECC tool, so that you can change a private key password, without changing its content, nor re-publish its associated .public key:

```
> ecc rekey
Enter the first chars of the .private certificate file name.
Auth: 8
Will use: 8BC90201EF55EE34F62DBA8FE8CF14DC.private

Enter the PassPhrase of this .private file.
AuthPass: #weLHn5E.Qfe
Enter the PassPhrase iteration rounds of this .private file.
AuthRounds [60000] :

Enter a NEW private PassPhrase for the key (at least 8 chars long).
Save this in a safe place: if you forget it, the key will be useless!
NewPass [mPy3kJWHE@LK] :
Corresponding TSynPersistentWithPassword.ComputePassword:
f+Gk8GCq1CA8Go

Enter the NEW PassPhrase iteration round for the key (at least 1000).
The higher, the safer, but will demand more computation time.
NewRounds [60000] :

8BC90201EF55EE34F62DBA8FE8CF14DC.private file created.
```

From now on, the root certificate will expect mPy3kJWHE@LK as keyphrase, for accessing its .private content. For instance (using only command line switches including the -noprompt option), you can now write:

```
> ecc infopriv -auth 8B -pass mPy3kJWHE@LK -noprompt
```
"Version": 1,
"Serial": "8BC90201EF55EE34F62DBA8FE8CF14DC",
"Issuer": "arbou",
"IssueDate": "2016-09-23",
"ValidityStart": "2016-09-23",
"ValidityEnd": "2017-09-23",
"AuthoritySerial": "8BC90201EF55EE34F62DBA8FE8CF14DC",
"AuthorityIssuer": "arbou",
"IsSelfSigned": true,
"Base64": "AQA1ADUAogG...."
}

Here, the "Base64": field only contains the public key information, not the private key content, which is kept secret and never serialized as JSON.

### 23.2.4. TECCCertificate and TECCCertificateSecret

As reference, here is how creating a new certificate is implemented in the ECC tool, and its .private/.public files generated, using TECCCertificateSecret class:

```pascal
function ECCCommandNew(const AuthPrivKey: TFileName;
const AuthPassword: RawUTF8; AuthPasswordRounds: integer;
const Issuer: RawUTF8; StartDate: TDateTime; ExpirationDays: integer;
const SavePassword: RawUTF8; SavePassordRounds, SplitFiles: integer): TFileName;
var
auth,new: TECCCertificateSecret;
begin
if AuthPrivKey=''
then auth := nil
else auth := TECCCertificateSecret.CreateFromSecureFile(AuthPrivKey,AuthPassword,AuthPasswordRounds);
try
// generate pair
new := TECCCertificateSecret.CreateNew(auth,Issuer,ExpirationDays,StartDate);
try
// save private key as .private password-protected binary file
new.SaveToSecureFiles(SavePassword,'.',SplitFiles,64,SavePassordRounds);
// save public key as .public JSON file
result := ChangeFileExt(new.SaveToSecureFileName,ECCCERTIFICATEPUBLIC_FILEEXT);
ObjectToJSONFile(new,result);
finally
new.Free;
e nd;
finally
auth.Free;
e nd;
end;
end;
end;
end;
end;
end;
end;
```

See the SynEcc.pas unit API reference, especially the TECCCertificateChain and TECCCertificateChainFile classes, which allow to store a certificate chain as JSON files or as a JSON array of base-64 encoded strings in your settings, using these constructors:

```pascal
constructor CreateFromJson(const json: RawUTF8);
constructor CreateFromArray(const values: TRawUTF8DynArray);
```

You can use the "source" command of the ECC tool to generate some pascal constant source code, containing an encrypted private key, ready to be embedded to your executable. For instance:

```bash
ecc source -auth 8 -pass mPy3kJWHE@LK -const MY_PRIV -noprompt
Will use: 8BC90201EF55EE34F62DBA8FE8CF14DC.private

8BC90201EF55EE34F62DBA8FE8CF14DC.private.inc file created.
```

When you look at the .private.inc generated file, you can directly use it in your source code, via copy and paste:

```pascal
const
With these classes, you have everything needed to implement your own private and secure PKI logic in your client/server applications.
In addition to some general information (name, date, size), you have unsigned hashes ("md5" and "sha256"), and an ECC digital signature, stored as a base-64 encoded string in the "sign": field. This signature has been computed using the 8BC90201EF55EE34F62DBA8FE8CF14DC.private key, and the SHA256 hash of the test1.txt file content. Note that you can add whatever JSON field you need to any .sign file, especially in the "meta": nested object, as soon as you don't modify the size/md5/sha256/sign values.

To verify the file, ensure that both test1.txt and test1.txt.sign files are in the current directory, then run:

```
> ecc verify -file test1.txt
test1.txt file verified as valid self signed.
```

Since the 8BC90201EF55EE34F62DBA8FE8CF14DC.private key has been signed using itself as authority, it is reported as "valid self signed". A signature verified against a certificate itself issued from another authority would have returned "valid signed".

Now if you modify test1.txt, e.g. changing one character, the verification will fail:

```
> ecc verify -file test1.txt
test1.txt file verification failure: invalid signature (9).
```

Don't forget to fix the test1.txt content back, since we will use it now as encryption source.

To check that you reverted to the original file content, run:

```
> ecc verify -file test1.txt
test1.txt file verified as valid self signed.
```

### 23.2.6. Signing in Code

From the source code point of view, you can easily add asymmetric digital signatures in your project using the TECCCertificateSecret.SignFile method, or working with memory buffer instead of files thanks to TECCCertificateSecret.SignToBase64 overloaded methods.

As reference, here is how the signing is implemented in the ECC tool:

```pascal
function ECCCommandSignFile(const FileToSign, AuthPrivKey: TFileName; const AuthPassword: RawUTF8; AuthPasswordRounds: integer): TFileName;
var
    auth: TECCCertificateSecret;
begin
    auth := TECCCertificateSecret.CreateFromSecureFile(AuthPrivKey, AuthPassword, AuthPasswordRounds);
    try
        result := auth.SignFile(FileToSign,[]);
    finally
        auth.Free;
    end;
end;
```

Verification can be done via the dedicated TECCSignatureCertified class:

```pascal
function ECCCommandVerifyFile(const FileToVerify, AuthPubKey: TFileName; const AuthBase64: RawUTF8): TECCValidity;
var
    content: RawByteString;
    cert: TECCSignatureCertified;
begin
    content := StringFromFile(FileToVerify);
    if content='' then
        raise EECCException.CreateUTF8('File not found: %', [FileToVerify]);
end;
```
cert := TECCSignatureCertified.CreateFromFile(FileToVerify);
try
  if not cert.Check then begin
    result := ecvInvalidSignature;
    exit;
  end;
  auth := TECCCertificate.Create;
try
  if auth.FromAuth(AuthPubKey, AuthBase64, cert.AuthoritySerial) then
    result := cert.Verify(auth, pointer(content), length(content))
  else
    result := ecvUnknownAuthority;
finally
  auth.Free;
end;
finally
  cert.Free;
end;

Here, the signing authority is supplied as a single .public local file, loaded in a TECCCertificate instance, but your projects may use TECCCertificateChain for a full PKI authority chain.

### 23.2.7. File Encryption

In order to encrypt out both test files, as proposed in Asymmetric Encryption Scheme (page 574), we will run the following commands:

```plaintext
$ ecc crypt -file test1.txt -auth 03 -saltpass monsecret -noprompt
Will use: 0388865C6B982A39E9EFB1DC1A95D227.public

test1.txt.synecc file created.

$ ecc crypt -file test2.txt -auth 03 -saltpass monsecret2 -noprompt
Will use: 0388865C6B982A39E9EFB1DC1A95D227.public

test2.txt.synecc file created.
```

As we can see, two new .synecc encrypted files have been computed:

```plaintext
$ dir test*
24/09/2016  16:36            94 161 test1.txt
24/09/2016  16:24               369 test1.txt.sign
24/09/2016  17:13            22 436 test1.txt.synecc
05/09/2016  10:37            72 209 test2.txt
24/09/2016  17:13            15 220 test2.txt.synecc
```

You may notice that the .synecc files are smaller than the original .txt files... in fact, SynEcc did recognize that the plain content was easily compressible, then applied SynLZ compression on it, before the encryption step.

If we ask for information about the test1.txt.synecc file:

```plaintext
$ ecc infocrypt -file test1.txt.synecc
{
  "Date": "2016-09-24",
  "Size": 94161,
  "Recipient": "toto",
  "RecipientSerial": "0388865C6B982A39E9EFB1DC1A95D227",
  "FileTime": "2016-09-24T16:36:59",
  "Algorithm": "ecaPKDF2_SHA256_AES256_CFB_SYNBLZ",
  "RandomPublicKey": "03C8320732D421D9A7F2F76C56562C79B97D13064EF2D6FAA20B0F246CD9B173890",
  "HMAC": "d6bf5679e58c68c45edf1fbd6a3b2e4d3e95e522f41e01ebadecab41f53183b",
  "Signature": {
    "Version": 1,
    "Date": "2016-09-24",
  }
```
We can see the information stored in the file header, including the recipient name and .publickey identifier, and also the "PBKDF2_HMAC_SHA256_AES256_CFB_SYNLZ" algorithm, which indeed includes _SYNLZ compression. Other algorithms are available (with diverse AES chaining modes), and some new methods may be added in the future.

The ecc crypt command did also include the digital signature available in the test1.txt.sign file in the current folder - so was in fact following Asymmetric Sign-Then-Encrypt Scheme (page 575) - whereas test2.txt.synecc does not have any embedded signature, since there was no test2.txt.sign file available at encryption time:

```bash
> ecc infocrypt -file test2.txt.synecc
{
    "Date": "2016-09-24",
    "Size": 72209,
    "Recipient": "toto",
    "RecipientSerial": "03B8865C6B982A39E9EF81DC1A95D227",
    "FileTime": "2016-09-05T10:37:50",
    "Algorithm": "ecaPBKDF2_HMAC_SHA256_AES256_CFB_SYNLZ",
    "RandomPublicKey": "03914AA67BCC92F78A9D670B2269587E3C97A3E08FD39A38845955811F88F7651",
    "HMAC": "07ddc90f7695fff8ca0683be98f7b1043e13a7bce879f8d6a928b6c5d9767d1",
    "Signature": "null"
}
```

As you can see, encryption is defined by its "Algorithm": field, and uses two additional properties:
- "RandomPublicKey" which contains a genuine key generated by ecc crypt, allowing perfect forward secrecy, meaning that a shared secret key is computed for every encryption: if someone achieves to break the AES256-CFB secret key used to encrypt a particular .synecc file (e.g. spending lots of money in brute force search), this secret key won't be reusable for any other file: each "RandomPublicKey" value above is indeed unique for each .synecc file;
- "HMAC": which uses a safe way of message authentication - known as keyed-hash message authentication code (HMAC) - stronger than the hashing algorithm it is based on, i.e. SHA256 in our case.

In practice, SynEcc implements state-of-the-art Elliptic Curve Integrated Encryption Scheme (ECIES) using PBKDF2_HMAC_SHA256 as key derivation function, AES256-CFB as symmetric encryption scheme, and HMAC-SHA256 algorithm for message authentication.


ECIES provides semantic security against an adversary who is allowed to use chosen-plaintext and chosen-ciphertext attacks. In addition to the expected genuine secret and message authentication in "RandomPublicKey" and "HMAC" properties, SynEcc implementation allows to customize the default "salt" value, to add a password protection for each .synecc encrypted file.

Decryption is pretty straightforward:

```bash
> ecc decrypt -file test1.txt.synecc
Enter the name of the decrypted file
Out [test1.txt.2] :

Enter the PassPhrase of the associated .private file.
AuthPass: b3dEB+DW8BJd

Enter the PassPhrase iteration rounds of this .private file.
AuthRounds [60000] :
```
Enter the optional PassPhrase to be used for decryption.
SaltPass [salt] : monsecret

Enter the PassPhrase iteration rounds.
SaltRounds [60000] :

    test1.txt.2 file verified as valid self signed.
    test1.txt.synecc file decrypted with signature.
    test1.txt.2 file created.

To decrypt the second file in a single step, and no console interaction:

```
> ecc decrypt -file test2.txt.synecc -authpass b3dEB+DW8BJd -saltpass monsecret2 -noprompt
```

    test2.txt.synecc file decrypted.
    test2.txt.2 file created.

As expected, the second file didn't contain any digital signature, so there is no "test2.txt.2 file verified as valid self signed." message.

The decrypted files are available in the current folder:

```
> dir test*
24/09/2016  16:36            94 161 test1.txt
24/09/2016  16:36            94 161 test1.txt.2
24/09/2016  16:24               369 test1.txt.sign
24/09/2016  17:13            22 436 test1.txt.synecc
05/09/2016  10:37            72 209 test2.txt
05/09/2016  10:37            72 209 test2.txt.2
24/09/2016  17:13            15 220 test2.txt.synecc
```

The *.2 decrypted files have the expect size (and content), after decompression. Even the file timestamp has been set to match the original.

### 23.2.8. Private Keys Passwords Cheat Mode

In order to follow best practice, our .private key files are always protected by a password. A random value with enough length and entropy is always proposed by the ECC tool when a key pair is generated, and could be used directly. It is always preferred to trust a computer to create true randomness (and SynCrypto.pas's secure TAESPRNG was designed to be the best possible seed, using hardware entropy if available), than using our human brain, which could be defeated by dictionary-based password attacks. Brute force cracking would be almost impossible, since PBKDF2_HMAC_SHA256 Password-Based Key Derivation Function with 60,000 rounds is used, so rainbow tables (i.e. pre-computed passwords list) will be inoperative, and each password trial would take more time than with a regular Key Derivation Function.

The issue with strong passwords is that they are difficult to remember. If you use not pure random passwords, but some easier to remember values with good entropy, you may try some tools like https://xkpasswd.net/... which returns values like $$$19*wrong*DRIVE*read*61$$$. But even then, you will be able to remember only a dozen of such passwords. In a typical public key infrastructure, you may create hundreds of keys, so remembering all passwords is no option for an average human being as you and me.

At the end, you end up with using a tool to store all your passwords (last trend is to use an online service with browser integration), or - admit it - store them in an Excel1 document protected by a password. Most IT people - and even security specialists - end with using such a mean of storage, just because they need it.

The weaknesses of such solutions can be listed:
- How could we trust closed source software and third-party online services?
- Even open source like http://keepass.info/help/base/security.html, may appear weak (no PBKDF, no AFSplit, managed C#, SHA as PRNG);
- The storage is as safe as the "master password" is safe;
- If the "master password" is compromised, all your passwords are published;
- You need to know the master password to add a new item to the store.

The ECC tool is able to work in "cheat mode", storing all .private key files generated passwords in an associated .cheat local file, encrypted using a cheat.public key.

As a result:
- Each key pair will have its own associated .cheat file, so you only unleash one key at a time;
- The .cheat file content is meaningless without the cheat.private key and its master password, so you can manage and store them together with your .private files;
- Only the cheat.public key is needed when creating a key pair, so you won't leak your master password, and even could generate keys in an automated way, on a distant server;
- The cheat.private key will be safely stored in a separated place, only needed when you need to recover a password;
- It uses strong File Encryption (page 585), with proven PBKDF, AFSplit, AES-PRNG, and ECDH/ECIES algorithms.

By default, no .cheat files are created. You need to explicitly initialize the "cheat mode", by creating master cheat.public and cheat.private key files:

```
> ecc cheatinit
Enter Issuer identifier text of the master cheat keys. Will be truncated to 15-20 ascii-7 chars.
Issuer [arbou] :

Enter a private PassPhrase for the master cheat.private key (at least 8 chars). Save this in a safe place: if you forget it, the key will be useless!
NewPass [uQHH*am39LLj] : verysafelongpassword

Enter iteration rounds for the mastercheat.private key (at least 100000).
NewRounds [100000] :

.cheat.public/.private file created.
```

As you can see, the default number of PBKDF rounds is high (100000), and local files have been created:

```
> dir cheat.*
18/10/2016  11:12             4 368 cheat.private
18/10/2016  11:12               568 cheat.public
```

Now we will create a new key pair (in a single command line, with no console interaction):

```
> ecc new -newpass NewKeyP@ssw0rd -noprompt

Corresponding TSynPersistentWithPassword.ComputePassword:
  encryption HeOyjDUAsOhvLZkMA0Y=
  authMutual lO0mv+8VpoFrrFfbBF11NppniWuma1L+AN3JXEUUpCY=
  authServer lO0nv+8VpoFrrFfbBF11NppniWuma1L+AN3JXEUUpCY=
  authClient lO0kv+8VpoFrrFfbBF11NppniWuma1L+AN3JXEUUpCY=

D1045FCBAA1382EE44ED2C212596E9E1.public/.private file created.
```
An associated .cheat file has been created:

```plaintext
>dir D10*

18/10/2016  11:15             1 668 D1045FCBAA1382EE44ED2C212596E9E1.cheat
18/10/2016  11:15             2 320 D1045FCBAA1382EE44ED2C212596E9E1.private
18/10/2016  11:15               588 D1045FCBAA1382EE44ED2C212596E9E1.public
```

Imagine you forgot about the NewKeyP@ssw0rd value. You could use the following command to retrieve it:

```plaintext
>ecc cheat

Enter the first chars of the .private certificate file name.
Auth: D10
Will use: D1045FCBAA1382EE44ED2C212596E9E1.private

Enter the PassPhrase of the master cheat.private file.
AuthPass: verysafelongpassword

Enter the PassPhrase iteration rounds of the cheat.private file.
AuthRounds [100000] :

```
{
    "pass": "NewKeyP@ssw0rd",
    "rounds": 6000
}
```

Corresponding TSynPersistentWithPassword.ComputePassword:
- encryption HeOyjDUAsOhvLZkMA0Y=
- authMutual 100mv+8VpoFrrFfbBFi1Npnn1WumaIL+AN3JXEUUpCY=
- authServer 100nv+8VpoFrrFfbBFi1Npnn1WumaIL+AN3JXEUUpCY=
- authClient 100kv+8VpoFrrFfbBFi1Npnn1WumaIL+AN3JXEUUpCY=

If your .private key does not have its associated .cheat file, you won't be able to recover your password:

```plaintext
>ecc cheat

Enter the first chars of the .private certificate file name.
Auth: BBC9
Will use: BBC90201EF55EE34F62DBA8FE8CF14DC.private

Enter the PassPhrase of the master cheat.private file.
AuthPass: verysafelongpassword

Enter the PassPhrase iteration rounds of the cheat.private file.
AuthRounds [100000] :

```
Fatal exception EECCException raised with message:
Unknown file BBC90201EF55EE34F62DBA8FE8CF14DC.cheat
```

In practice, this "cheat mode" will help you implement a safe public key infrastructure of any size. It will be as secure as the main cheat.private key file and its associated password remain hidden and only wisely spread, of course. Don't forget to use the ecc rekey command on a regular basis, so that you change the master password of cheat.private. The main benefit of this implementation is that for all key generation process, only the cheat.public key file is needed.
23.2.9. Encryption in Code


As reference, here is how the encryption is implemented in the ECC tool:

```pascal
procedure ECCCommandCryptFile(const FileToCrypt, DestFile, AuthPubKey: TFileName;
const AuthBase64, AuthSerial, Password: RawUTF8; PasswordRounds: integer; Algo: TECIESAlgo);
var
  content: RawByteString;
  auth: TECCCertificate;
begin
  content := StringFromFile(FileToCrypt);
  if content='' then raise EECException.CreateUTF8('File not found: %', [FileToCrypt]);
  auth := TECCCertificate.Create;
  try
    if auth.FromAuth(AuthPubKey, AuthBase64, AuthSerial) then begin
      auth.EncryptFile(FileToCrypt, DestFile, Password, PasswordRounds, Algo, true);
    end;
  finally
    auth.Free;
    FillZero(content);
  end;
end;
```

You may note here the use of FillZero() in the finally block of the function, which is a common - and strongly encouraged - way of protecting your sensitive data from remaining in RAM, after use. Both SynCrypto.pas and SynEcc.pas code has been checked to follow similar safety patterns, and not leave any sensitive information in the program stack or heap.

23.3. Application Locking

A common feature request for professional software is to prevent abuse of published applications. For licensing or security reasons, you may be requested to "lock" the execution of programs, maybe tools or services.

mORMot can use Asymmetric Cryptography to ensure that only allowed users could run some executables, optionally with dedicated settings, on a given computer. The framework offers the first brick, on which you should build upon your dedicated system.

The dddInfraApps.pas unit publishes the following ECCAuthorize function and type:

```pascal
type
  TECCAuthorize = (eaSuccess, eaInvalidSecret, eaMissingUnlockFile,
                   eaInvalidUnlockFile, eaInvalidJson);

function ECCAuthorize(aContent: TObject; aSecretDays: integer; const aSecretPass,
                       aDPAPI, aDecryptSalt, aAppLockPublic64: RawUTF8; const aSearchFolder: TFileName = '';
                       aSecretInfo: PECCCertificateSigned = nil; aLocalFile: PFileName = nil): TECCAuthorize;
```

This function will use several asymmetric key sets:
- A **main key set**, named e.g. applock.public and applock.private, shared for all users of the system;
- Several **user-specific key sets**, named e.g. user@host.public and user@host.secret, one for each user and associated computer host name.

When the ECCAuthorize function is executed, it will search for a local user@host.unlock file, named
after the current logged user and the computer host name. Of course, the first time the application is launched for this user, there will be no such file. It will create two local user@host.public and user@host.secret files and return eaMissingUnlockFile.

The main key set will be used to digitally sign the unlock file:
- applock.public will be supplied as plain base64-encoded aAppLockPublic64 text parameter in the executables - for safety, you should ensure its value is not replaced by a forged one by an attacker: the executable should be signed, or at least the constant value should be checked with a CRC for its content during the program execution;
- On the contrary, applock.private will be kept secret - with its associated secret password.

User-specific key sets will be used to encrypt the unlock file:
- The user@host.secret file contains in fact a genuine private key, encrypted using CryptDataForCurrentUser (i.e. DPAPI under Windows) for the specific computer and user: this will avoid user@host.unlock reuse on another computer, even if the user and host names are identical, and the user@host.secret file is copied. This file should remain local, and doesn't need to be transmitted.
- The user@host.public file will be sent to the product support team, e.g. by email - but you may setup an automated server, if needed. The support team will create a user@host.unlock matching this user@host.public key, which will unlock the application for the given user.

On the support team side, a user@host.json file is created for the given user, and will contain the JSON serialization of the aContent: TObject parameter of the ECCAuthorize function. This object may contain any published properties, matching the security expectations for this user, e.g. the available features or resource access.

23.3.1. From the User perspective
The resulting process is therefore the following:

Application Unlocking via Asymmetric Cryptography

In short, every user/computer combination will have its own set of public/secret/unlock files.
- In practice, applock.public could be hardcoded as plain base64-encoded aAppLockPublic64
constant string in the Application code - of course, the executable should be signed with a proper authority, to ensure this constant is not replaced by a fake value;
- The location of those local user@host.* files is by default the executable folder, but may be specified via the aSearchFolder parameter - especially if this folder is read-only (e.g. due to Windows UAC), or if you use some custom GUI for the user interactivity;
- The user@host.json will be signed using applock.private secret key, to testify that the resulting user@host.unlock file was indeed provided by the Support Team;
- The user@host.json will be encrypted using the user@host.public key received by email, so will be specific to a single user/computer combination.

If two users share the application on the very same computer, another set of files will appear:

![Application Unlocking on Two Computers](image)

Several users on the same computer will be handled as such:

![Application Unlocking for Two Users](image)

From the User point of view, he/she will transmit its user@host.public file, then receives a corresponding user@host.unlock file, which will unlock the application. Pretty easy to understand - even if some complex asymmetric encryption is involved behind the scene.

### 23.3.2. From the Support Team perspective

The Support Team will maintain a list of user@host.public and user@host.json files, one per user/computer. Both files have small JSON content, so may be stored in a dedicated folder of the project source code repository - or in a dedicated repository. The use of a source code repository allows to track user management information between several support people, including history and audit trail of this sensitive information. For safety, the applock.private file may not be archived in the source code repository, but copied on purpose on each support people's (or developer's) computer. A separated, and dedicated computer, may be used, for additional safety.
In fact, even developers may define their own set of .unlock files. For local test builds, they may use their own applock.public and applock.private key pairs, diverse from the main content.

The content of each user@host.json may be easily derived from a set of reference .json files, acting like templates of group of users. Or an existing file may be used as source for a new user. The ability to use JSON and a text editor, with customizable object and arrays fields, allows any needed kind of licensing or security scope, depending on the application.

Since the user@host.json is a serialized aContent: TObject, you can define enumerates properties, or even schema-less structures as TDocVariant - see TDocVariant custom variant type (page 112) - to refine the authorization scope.

The user@host.json file is encrypted using the genuine user@host.public key, and its associated user@host.secret is strongly encrypted for the given PC and logged user: therefore, only the application is able to decipher the user@host.unlock content. You can let those files be transmitted via an unsafe mean of transport, e.g. plain email, with no compromising risk. Last but not least, passwords or IP addresses can be safely stored in its content, as part of the security policy of your project.

In practice, the team may use a unlock.bat file running the ECC tool over secret applock.private keys, containing the secret:

```batch
@echo off
echo Usage:  unlock user@host
echo.
ecc sign -file %1.json -auth applock -pass applockprivatepassword -rounds 60000
ecc crypt -file %1.json -out %1.unlock -auth %1 -saltpass decryptsalt -saltrounds 10000
del %1.json.sign
```

For safety, you may not include the -pass applockprivatepassword value in this unlock.bat file. Removing this -pass command-line switch will let the ecc tool prompt for the password secret key on the console:

```bash
ecc sign -file %1.json -auth applock -rounds 60000
```

Also note that you can use the ecc rekey command to customize the password of a given applock.private file: each support team member may have his/her custom password to run the sign-then-encrypt process.

Of course, if you need to create a lot of .unlock files, you may want to automate this process, e.g. in a server or a GUI tool, using SynEcc.pas classes.

### 23.3.3. Benefits of Asymmetric Encryption for License management

In most licensing systems, the weak point is the transmission of the licensing file. Thanks to Asymmetric Encryption, both user@host.public and user@host.unlock files can be transmitted as plain emails, without any possibility of compromising.

The applock.private secret key and its associated password are used to digitally sign (using ECDSA) the plain content of the user@host.unlock file. This sign-then-encrypt pattern will ensure that only your support team will be able to generate the proper .unlock files for a given application. The applock.private/public keys could have their own deprecation date.

As we have seen, the user@host.unlock file is encrypted, so you can use it to transmit sensitive information. Its associated user@host.secret key has been generated locally with an expiration date - see the aSecretDays parameter of the ECCAuthorize function. It will ensure that the registering process should be performed regularly, if the licensing or security policy expect it.
Of course, any such system is as weak as its weakest point. In particular, under Windows the executable should be digitally signed (as any professional software). You could also ensure that the aAppLockPublic64 public key has not been replaced by a fake value forged by an attacker - e.g. by checking its value by computing its CRC in several places of your application:

```pascal
if crc32($1239438,pointer(AppLock64),length(AppLock64))<>$ae293c10 then Close;
```

The security of this system does not rely on code obfuscation, but on proven safety of asymmetric encryption. Even if the executable is modified in-place to by-pass the license check, the fact that the application expects some additional information to be provided within the user@host.unlock file will make it much more difficult to hack.

As always with Open Source, any feedback is welcome, in order to enhance the safety of this system. The fact that the code is available - so that the algorithms could be proven - make it safer than any proprietary solution developed in-door.
24. Domain-Driven-Design

We have now discovered how mORMot offers you some technical bricks to play with, but it is up to you to build the house (castle?), according to your customer needs.

This is were Domain-Driven Design (page 99) - abbreviated DDD - patterns are worth looking at.

24.1. Domain

What do we call Domain here?

The domain represents a sphere of knowledge, influence or activity.

As we already stated above, the domain has to be clearly identified, and your software is expected to solve a set of problems related to this domain.

DDD is some special case of Model-Driven Design. Its purpose is to create a model of a given domain. The code itself will express the model: as a consequence, any code refactoring means changing the model, and vice-versa.
24.2. Modeling

Even the brightest programmer will never be able to convert a real-life domain into its software code. What we can do is to create an abstraction system that describes selected aspects of a domain.

Modeling is about filtering the reality, for a given use context: "All models are wrong, some are useful" G. Box, statistician.

24.2.1. Several Models to rule them all

As first consequence, several models may coexist for a given reality, depending of the knowledge level involved - what we call a Bounded Context. Don't be afraid if the same reality is defined several times in your domain code: you should use only one class in a given context, but you may have another class defined in another context, with diverse attributes or methods.

Just open Google maps for instance, and think how the same reality may be modeled depending on the zoom level, or you current view options. See also the M1, M2, M3 models as defined in Meta-Object Facility. When you define several models, you just need to clearly state the current model you are using.

Even models could be abstracted. This is what DDD does: the code itself is some kind of meta-model, conforming a given conceptual model to the grammar of a given programming language.

24.2.2. The state of the model

Most models express the reality in two dimensions:
- Static: to abstract a given state of the reality;
- Dynamic: to abstract how reality evolves (i.e. its behavior).

In both dimensions, we can clearly understand the purpose of abstraction.

Since it is impossible to model all the details of reality (e.g. describe a physical reality down to atomic / sub-atomic level), the static modeling will forget the non significant details, and focus on the essentials, for a given knowledge level, which is specific to a given context.

Similarly, most changes are continuous in the world, but dynamic modeling will create static snapshots of the reality (called state transitions), to embrace the deterministic nature of computers.

State always brings complexity to the model. As a consequence, our code should be as stateless as possible.

Therefore:
- Try to always separate value and time in state;
- Reduce statefulness to the only necessary;
- Implement your logic as state machines instead of blocking code or sessions;
- Persistence should handle one-way transactions.

In DDD, Value Objects and Entity Objects are the mean to express a given system state. Immutable Value Objects define a static value. Entity refers to a given state of given identity (or reality).

For instance, the same identity (named "John Doe") may be, at a given state, single and minor, then, at another state, married and adult. The model will help to express the given states, and the state transitions between them (e.g. John's marriage).

In DDD, the Factory / Repository / Unit Of Work patterns will introduce transactional support in a stateless approach.
And in situations where a reality does change its state very often, with complex impacts on other components, DDD will model these state changes as Events. It could lead into introducing some Event-Driven Design even or Event Sourcing within the global model.

### 24.2.3. Composition

In order to refine your model, you have two main tools at hand to express the model modularity:

- **Partitioning**: the more your elements have a separated concern, the better;
- **Grouping**: to express constraints, elements may be grouped - but usually, you should not put more than 6 or 8 elements in the same diagram, or your model may need to be refined.

In DDD, a lot of small objects have to be defined, in order to properly partition the logic. When we start with Object Oriented Programming, we are tempted to create huge classes with a lot of methods and parameters. This is a symptom of a weak model. We should always favor composition of small simple objects, just like the Unix tools philosophy or the Single Responsibility Principle - see SOLID design principles (page 389).

Some DDD experts also do not favor inheritance. In fact, inheriting may be also a symptom of some coupled context. Having two diverse realities sharing properties may be a bad design smell: if two or more classes inherit from one parent class, the state and behavior of the parent class may limit any future evolution of any of its children. In practice, trying to follow the Open/Closed Principle - see Open/Closed Principle (page 393) - at class level may induce unexpected complexity, therefore reducing code maintainability.

In DDD, the **Aggregate Root** is how you group your objects, in order to let constraints (e.g. business rules) to be modeled. Aggregates are the main entry point to the domain, since they should contain, by design, the whole execution context of a given process. Their extent may vary during development, e.g. when a business rule evolves - remember that the same reality can appear several times in the same domain, but once per Bounded Context. In other words, Aggregates could be seen as the smallest and biggest extent needed to express a given model context.
24.3. DDD model

It is now time to define which kind of Model-Driven Design is DDD:

24.3.1. Ubiquitous Language

Ubiquitous Language is where DDD begins.

DDD expects the domain model to be expressed via a shared language, and used by all team members to connect their activities with the software. Those terms should be used in speech, writing, and any presentation or diagram.

In the real outside world, i.e. for the other 10th kind of people how do not know about binary, domain experts use company- or industry-standard terminology.

As developers, we have to understand this vocabulary and not only use it when speaking with domain experts but also see the same terminology reflected in our code. If the terms "class code" or "rate sets" or "exposure" are frequently used in conversation, we shall find corresponding class names in the code. In DDD, it is critical that developers use the business language in code consciously and as a disciplined rule. As a consequence, browsing the code should lead into a clear comprehension of the business model.

Domain experts will be the guard keepers of the consistency of this language, and its proper definition. Even if the terms are expected to be consistent, they are not to be written in stone, especially during the initial phase of software development. As soon as one domain activity cannot be expressed using the existing set of concepts, the model needs to be extended. Removing ambiguities and inconsistencies is a need, and will, very often, resolve several not-yet-identified software issues.

24.3.2. Value Objects and Entities

For the definition of your objects or internal data structures (what good programmers care about), you are encouraged to make a difference between several kind of objects. Following DDD, model-level representation are, generally speaking, rich on behavior, therefore also of several families/species of objects.

Let us list the most high-level definitions of objects involved to define our DDD model:
- **Value Objects** contain attributes (value, size) but no conceptual identity - e.g. money bills, or seats in a Rock concert, as they are interchangeable;

- **Entity objects** are not defined by their attributes (values), but by their **thread of continuity**, signified by an identity - e.g. persons, or seats in most planes, as each one is unique and identified.

The main difference between Value Objects and Entities is that instances of the second type are tied to one reality, which evolves in the time, therefore creating a thread of continuity.

Value objects are immutable by definition, so should be handled as read-only. In other words, they are incapable of change once they are created.

Why is it important that they be immutable? With Value objects, you're seeking side-effect-free functions, yet another concept borrowed by DDD to functional languages (and not available in most OOP languages, until latest concurrent object definition like in Rust or Immutable Collections introduced in C#/.NET 4.5). When you add $10 to $20, are you changing $20? No, you are creating a new money descriptor of $30. A similar behavior should be visible at code level.

Entities will very likely have an ID field, able to identify a given reality, and model the so-called thread of continuity of this identity. But this ID is an implementation detail, only used at Persistence Layer level: at the Domain Layer level, you should not access Entities individually, but via a special Entity bounded to a specific context, called Aggregate Root (see next paragraph).

When we define some objects, we should focus on making the implicit become explicit. For instance, if we have to store a phone number, we won't use a plain string type for it, but we will create a dedicated Value object type, making explicit all the behavior of its associated reality. Then we will be free to combine all types into explicit grouped types, on need.

### 24.3.3. Aggregates

Aggregates are a particular case of Entities, defined as collection of objects (nested Values and/or Entities) that are grouped together by a root Entity, otherwise known as an Aggregate Root, which scope has been defined by a given execution context - see "Composition" above.

Typically, Aggregates are persisted in a database, and guarantee the consistency of changes by isolating its members from external objects (i.e. you can link to an aggregate via its ID, but you can not directly access to its internal objects). See [Shared nothing architecture (or sharding)](http://martinfowler.com/bliki/AggregateOrientedDatabase.html) (page 155) which sounds like http://martinfowler.com/bliki/AggregateOrientedDatabase.html.

In practice, Aggregates may be the only kind of objects which will be persisted at the Application layer, before calling the domain methods: even if each nested Entity may have its own persistence method (e.g. one RDBMS table per Entity), Aggregates may be the unique access point to retrieve or update a given state. It will ensure so-called Persistence Ignorance, meaning that domain should remain uncoupled to any low-level storage implementation detail.

DDD services may just permit remote access to Aggregates methods, where the domain logic will be defined and isolated.

### 24.3.4. Factory and Repository patterns

DDD then favors some patterns to use those objects efficiently.

The **Factory pattern** is used to create object instances. In strongly-typed OOP (like in Delphi, Java or C#), this pattern is in fact its constructor method and associated class type definition, which will define a fixed set of properties and methods at compilation time (this is not the case e.g. in JavaScript or weak-typed script languages, in which you can add methods and properties at runtime).
In fact, Delphi is ahead of Java or C#, since it allows virtual constructors to be defined. Those virtual constructors are in fact a clean and efficient way of implementing a Factory, and also fulfill SOLID principles, especially the Liskov Substitution Principle (page 395): the parent class define an abstract constructor on which you rely, but the implementation will take place in the overridden constructor.

The Factory pattern can also be used to create interface instances - see Interfaces (page 385). Main benefit is that alternative implementations may be easily interchanged. Such abstraction helps testing - see Interfaces in practice: dependency injection, stubs and mocks (page 406) - but also introduces interface-based services - see Client-Server services via interfaces (page 419).

Repository pattern is used to save and dispense each Aggregate Root. It matches the "Layer Supertype" pattern (see above), e.g. via our mORMot TSQLRecord and TSQLRest classes and their Client-Server ORM features, or via dedicated repository classes - saving data is indeed a concern orthogonal to the model itself. DDD architects claim that persistence is infrastructure, not domain. You may benefit in defining your own repository interface, if the standard ORM / CRUD operations are not enough.

24.3.5. DTO and Events to avoid domain leaking

The main DDD architecture principle - and benefit - is to isolate the domain code. As will be defined by the Hexagonal architecture - see below (page 602), everything is made to ensure that the domain won't "leak" outside its core. The domain objects and services are the most precious part of any DDD project, especially in the long term, so proper isolation and uncoupling sound mandatory.

The Aggregates should always be isolated and stay at the Application layer, given access to its methods and nested objects via proper high-level remote Services - see below (page 601) - which should not be published directly to the outer world either.

In practice, if your domain is properly defined, most of your Value Objects may be sent to the outer world, without explicit translation. Even Entities may be transmitted directly, since their methods should not refer to nothing but their internal properties, so may be of some usefulness outside the domain itself.

But the real world may be rough and cruel, and optimism will better be replaced by some kind of pragmatism, and a pinch of cynicism. DDD experience told its pioneers (sometimes in a painful manner), that Adapters types should better be defined, especially at Application layer and Presentation layer levels.

As a result, a new family of objects will secure any DDD implementation:
- Data Transfer Objects (DTO) are transmission objects, which purpose is to not send your domain across the wire (i.e. separate your layers, following the Anti-Corruption Layer pattern). It encourages you to create gatekeepers (e.g. in the Application layer) that work to prevent non-domain concepts from leaking into your model.
- Commands and Events are some kind of DTO, since they communicate data about an event and they themselves encapsulate no behavior.

Using such dedicated types will eventually help uncoupling the domain, for several reasons:
- You can refactor your domain, without the need to modify the published interfaces, but just the tiny Anti-Corruption layer: no need for your customers to spend money upgrading their client applications, just because your domain changed; no fear to refine your precious domain code, in which you put all your money and expectations, just because it may be unpleasant to your customers.
- End-user application expectations won't pollute your domain. For instance, you will better define a per-customer set of public APIs, rather than exposing your domain services. In practice, a "one to rule them all" public API may sound like a good idea at first, but it will eventually end up as a monstrous, flat, unreadable and anemic interface, far away from SOLID design principles (page 389).

- Since the domain tends to be as generic as possible, its objects may sometimes be overkill to the end user applications: if some properties will never be used, or will always be void, why will you pollute your end user code, and waste bandwidth or resources? Just stick to what is needed.

- Dedicated types will help focusing on the needed use cases, so will ease documentation, maintainability, testing and integration with client applications: even translating your Ubiquitous language objects into more common or expected terms in the presentation layer will be beneficial.

- Consider that in your company, the Domain and Infrastructure layers may be maintained by your most valuable teams, whereas some less skilled developers (or even offshore teams) may be involved on Application and Presentation layers. Writing adapter/translator classes is not difficult, and will help your company focus and invest on where long term ROI is more likely to appear. Some access restrictions may therefore appear at source code level: it may be safe that only the wiser programmers will be allowed to modify the domain code, and even hide the domain implementation by publishing only its interfaces, protecting your most valuable intellectual property from being copied and stolen.

In mORMot, we try to let the framework do all the plumbing, letting those types be implemented via interfaces over simple dedicated types like records or dynamic arrays - see Service Methods Parameters (page 423) and Asynchronous callbacks (page 444). So defining DTOs, Commands and Events in dedicated Anti-Corruption layers will be pretty much quick, easy and safe.

### 24.3.6. Services

Aggregate roots (and sometimes Entities), with all their methods, often end up as state machines, and the behavior matches accordingly.

In the domain, since Aggregate roots are the only kind of entities to which your software may hold a reference, they tend to be the main access point of any process. It could be handy to publish their methods as stateless Services, isolated at Application layer level.

*Domain services pattern* is used to model primary operations.

Domain Services give you a tool for modeling processes that do not have an identity or life-cycle in your domain, that is, that are not linked to one aggregate root, perhaps none, or several. In this terminology, services are not tied to a particular person, place, or thing in my application, but tend to embody processes. They tend to be named after verbs or business activities that domain experts introduce into the so-called Ubiquitous Language. If you follow the interface segregation principle - see Interface Segregation Principle (page 400), your domain services should be exposed as dedicated client-oriented methods. Do not leak your domain! In DDD, you develop your Application layer services directly from the needs of your client applications, letting the Domain layer focus on the business logic.

*Unit Of Work* can be used to maintain a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems. In short, it implements transactional process at Domain level, and may be implemented either at service or ORM level. It features so-called Persistence Ignorance, meaning that your domain code may not be tied to a particular persistence implementation, but "hydrate" Aggregate roots class instances as abstractly as possible.

A dual-phase commit approach - with some methods preparing and validation the data, then applying it by a dedicated Commit command in a second step - may be defined. In this pattern, the repository is
just some simple storage, and data consistency will take place at domain level: for instance, you will not define any SQL constraints, but validate your data before storing the information. Your business rules should be written in high level domain code, and you may forget about the FOREIGN KEY, or CHECK SQL syntax flavors. As a result, you may safely change from a SQL database to a NoSQL engine, or even a TObjectList. You will be able to define and maintain any complex business rules, using the Ubiquitous Language of your domain. And a change of business logic will not impact the database metadata, which may be painful to modify.

The DDD Services may therefore be stateless for most of the time, but allowing some flavor of transactional process, when needed. The uppermost/peripheral architecture layers - i.e. Application or Presentation Layers - will ensure that those services will be properly orchestrated. The application workflows will not be defined in the domain core itself, but in those outer layers, resulting in a cleaner, uncoupled architecture.

24.3.7. Clean Uncoupled Architecture

If you follow properly the DDD patterns, your classic Multi-tier architecture (page 88) architecture will evolve into a so-called Clean Architecture or Hexagonal architecture.

Even if physically, this kind of architecture may still look like a classic layered design (with presentation on the top, business logic in the middle and a database at the bottom - and in this case we speak of N-Layered Domain-Oriented Architecture), DDD tries to isolate the Domain Model from any dependency, including technical details.

As a consequence, the logical architecture of any DDD solution should appear as such:
Clean Uncoupled Domain-Oriented Architecture

That kind of architecture is not designed in layers any more, but more like an Onion.

At the core of the bulb - sorry, of the system, you have the **Domain Model**. It implements all **Value Objects** and **Entity Objects**, including their state and behavior, and associated unit tests.

Around this core, you find **Domain Services** which add some more behavior to the inner model. Typically, you will find here abstract interfaces that provides persistence (**Aggregates** saving and retrieving via the **Repository** pattern), let **Domain objects properties and methods be defined** (via the **Factory** pattern), or access to third-party services (for service composition in a SOA world, or e.g. to send a notification email).

Then **Application Services** will define the **workflows** of all end-user applications. Even if the core Domain is to be as stable as possible, this outer layer is what will change more often,
depending on the applications consuming the Domain Services. Typically, workflows will consist in dehydrating some Aggregates via the Repository interface, then call the Domain logic (via its objects methods, or for primary operations with wider Domain services), call any external service, and validate ("commit", following Unit-Of-Work or transactional terms) objects modifications. Some non data-centric process will also benefit from a dual-phase commit pattern, to allow safe orchestration of uncoupled domain and third party services.

Out on the edges you see User Interface, Infrastructure (including e.g. database persistence), and Tests. This outer layer is separated from the other three internal layers, which are sometimes called Application Core.

This is where all technical particularities will be concentrated, e.g. where RDBMS / SQL / ORM mapping will be defined, or platform-specific code will reside. This is the right level to test your end-user workflows, e.g. using Behavior-Driven Development (abbreviated BDD), with the help of your Domain experts.

The premise of this Architecture is that it controls coupling. The main rule is that all coupling is toward the center: all code can depend on layers more central, but code cannot depend on layers further out from the core. This is clearly stated in the Clean Uncoupled Domain-Oriented Architecture (page 603) diagram: just follow the arrows, and you will find out the coupling order. This architecture is unashamedly biased toward object-oriented programming, and it puts objects before all others.

This Clean Architecture relies heavily on the Dependency Inversion principle - see SOLID design principles (page 389). It emphasizes the use of interfaces for behavior contracts, and it forces the externalization of infrastructure to dedicated implementation classes. The Application Core needs implementation of core interfaces, and if those implementing classes reside at the edges of the application, we need some mechanism for injecting that code at runtime so the application can do something useful. mORMot’s Client-Server services via interfaces (page 419) provide all needed process to access, even remotely, e.g. to persistence or any third party services, in an abstract way.

With Clean Architecture, the database is not the center of your logic, nor the bottom of your physical design - it is external. Externalizing the database can be quite a challenge for some people used to thinking about applications as "database applications", especially for Delphi programmers with a RAD / TDataSet background. With Clean Architecture, there are no database applications. There are applications that might use a database as a storage service but only though some external infrastructure code that implements an interface which makes sense to the application core. The domain could be even decoupled from any ORM pattern, if needed. Decoupling the application from the database, file system, third party services and all technical details lowers the cost of maintenance for the life of the application, and allows proper testing of the code, since all Domain interface types could be mocked on purpose - see Stubs and mocks (page 407).
24.4. mORMot's DDD

24.4.1. Designer's commitments
Before going a bit deeper into the low-level stuff, here are some key sentences we should better often refer to:
- I shall collaborate with domain experts;
- I shall focus on the ubiquitous language;
- I shall not care about technical stuff or framework, but about modeling the Domain;
- I shall make the implicit explicit;
- I shall use end-user scenarios to get real and concrete;
- I shall not be afraid of defining one model per context;
- I shall focus on my Core Domain;
- I shall let my Domain code uncoupled to any external influence;
- I shall separate values and time in state;
- I shall reduce statefulness to the only necessary;
- I shall always adapt my model as soon as possible, once it appears inadequate.

As a consequence, you will find in mORMot no magic powder to build your DDD, but all the tools you need to focus on your business, without loosing time in re-inventing the wheel, or fixing technical details.

24.4.2. Defining objects in Delphi
How to implement all those DDD concepts in an object-oriented language like Delphi? Let's go back to the basics. Objects are defined by a state, a behavior and an identity. A factory helps creating objects with the same state and behavior.

In Delphi and most Object-Oriented (OOP) languages - including C# or Java, each class instance has the following behavior:
- State is defined by all its property / member values;
- Behavior are defined by all its methods;
- Identity is defined by reference, i.e. a=b is true only if a and b refers to the same object;
- Factory is in fact the class type definition itself, which will force each instance to have the same members and methods.

In Delphi, the record type (and deprecated object type for older versions of the compiler) has an alternative behavior:
- State is also defined by all its property / member values;
- Behavior are also defined by all its methods;
- But identity is defined by content, i.e. RecordEqual(a, b) is true only if a and b have the same exact property values;
- Factory is in fact the record / object type definition itself, which will force each instance to have the same members and methods.

In practice, you may use either one of the two kinds of object types (i.e. either class or record), depending on the behavior expected by DDD patterns:
- DDD's DTO may be defined as record, and directly serialized as JSON via text-based Record serialization (page 297) - as an alternative, you may consider using TDocVariant custom variant type (page 112);
- But other kinds of DDD objects, i.e. Value Objects, Entity Objects and Aggregates, should better be defined as dedicated class, since class type definition offers more possibility than plain record structures. The framework defines some parent classes (e.g. TSynPersistent and TSynAutoCreateFields) which makes working with class instances almost as easy than stack-allocated record values.

24.4.3. Defining DDD objects in mORMot

When defining domain objects, we should always make the implicit explicit, i.e. writing one class type per reality in the model, in every bounded context. Thanks to Delphi's strong typing, you will ensure that the Domain Ubiquitous language will appear in the code, and that your model will be expressed in a clean, uncoupled way.

If those class types are defined as plain PODO, even your domain experts - which may not know anything about writing code - may be part of the class definition: we usually write the domain objects and services with the domain experts, writing the code in real time during a meeting. The domain is therefore expressed as plain code, and experts are able to validate the workflows and properties of the model as soon as possible. Such coding sessions truly benefit of being a cooperative team work, not only coders'.

Once the domain model is stabilized, we may start implementing the interfaces using this common work as contract. In this implementation process, the mORMot framework offers a lot of tools to make it happen in a quick and efficient manner.

There are in fact two ways of implementing DDD objects as class types, in mORMot:
- Directly using the framework types, e.g. TSQLRecord specialized class for Entities or Aggregates;
- Or relying of no framework structure, but clean PODOs (Plain Old Delphi Object - see so-called POJO or POCO for Java or C#) class types, then use the mORMotDDDD.pas unit for automatic marshalling.

Of course, the second option may be preferred, since it sounds like a better implementation path, uncoupled from the framework itself. Remember that DDD is mainly about uncoupling the Domain code from any external dependency, even from mORMot itself. You should better not be forced to use the framework ORM, if you have some existing legacy SQL statements, for instance.

24.4.3.1. Use framework types for DDD objects

If you want to directly use framework structure, DDD's Value Objects are probably meant to be defined as record, with methods (i.e. in this case as object for older versions of Delphi). You may also use TComponent or TSQLRecord classes, ensuring the published properties do not have setters but just read F... definition, to make them read-only, and, at the same time, directly serializable.

If you use record / object types, you may need to customize the JSON serialization - see Record serialization (page 297) - when targeting AJAX clients, especially for any version prior to Delphi 2010 (by default, records are serialized as binary + Base64 encoding due to the lack of enhanced RTTI, but you can define easily the record serialization e.g. from text). Note that since record / object defines in Delphi by-value types (whereas class defines by-reference types - see previous paragraph), they are probably the cleanest way of defining Value Objects.

In this context, DDD's Entity objects could inherit from TSQLRecord. It will give access to a whole set of methods supplied by mORMot, implementing some kind of "Layer Supertype", as explained by Martin Fowler.

Finally, DDD's Aggregates will benefit of using mORMot's Object-Relational Mapping (page 130). Entities will be stored as regular TSQLRecord, e.g. using "One to one" or "One to many" (page 151)
cardinality, as available from the framework.

For most simple cases, this solution may be just good enough. But it may have the drawback of coupling your *Domain logic* with *mORMot* internals. Your Domain will eventually be polluted by the framework implementation details, which should better be avoided.

### 24.4.3.2. Define uncoupled DDD objects

In order to uncouple our *Domain* code from its persistence layer, *mORMot* offers some dedicated types and units to use PODO class definitions within your DDD core.

You may use regular TPersistent as parent class, but you may consider using TSynPersistent and TSynAutoCreateFields fields instead - we will see soon their benefit.

Let's start from existing code, available in the SQLite3\DDD\dom sub-folder of the framework source code repository, in the dddDomUserTypes.pas unit. This unit defined some reusable class types, able to store user information, in a clean DDD way.

### 24.4.3.3. Specialize your simple types

Each reality in this unit will have its own type definition, using the extended pascal syntax, even for simple types like string or integer:

```pascal
type
  TSpecifiedType = type TParentType;
```

You may not be familiar with this syntax. But it is a pretty powerful mean of defining your DDD model with a plain pascal syntax. Here TSpecifiedType is defined as a specific type, which will behave like TParentType, but *strong-typing* will apply in your code, so that the compiler will complain if you pass e.g. a TParentType instead of a TSpecifiedType as a var parameter. It will help to resolve some ambiguities when transmitting information.

For instance, in the dddDomUserTypes.pas unit, you may see:

```pascal
  type
    TLastName = type RawUTF8;
    TFirstName = type RawUTF8;
    TMiddleName = type RawUTF8;
```

Thanks to those type definitions, you will be able to make a difference between a last name, a first name and a middle name. We used RawUTF8 as parent type, but we may have used string. Since we wanted our code to work seamlessly with all versions of Delphi and FPC, we rather rely on RawUTF8 - see *Unicode and UTF-8* (page 105).

Once compiled, there won't be any difference between the three types, which will behave like a RawUTF8. But at compile time, and in your Domain source code, you will be able to know exactly which reality is stored in a given variable.

So instead of this method definition:

```pascal
  function UserExists(const aUserName: RawUTF8): boolean;
```

You will rather write:

```pascal
  function UserExists(const aUserName: TLastName): boolean;
```

With such a method signature, we will ensure that we won't supply a TFirstName or a TPetName by mistake.

It may sound like a small enhancement, but be sure that it will increase your code safety, and
expressiveness. One of the biggest failure in NASA history was *Mars Climate Orbiter*. A variable type error burn up a $327.6 million project in minutes, when one engineering group working on the thrusters measured in English units of pounds-force seconds, whereas the others used metric Newton-seconds. The result of that inattention is now lost in space, possibly in pieces.

Remember when our physic teachers leaped all over answers that consisted of a number. If the answer was 2.5, they will take their red pens and write "2.5 what? Weeks? Puppies? Demerits?" And proceed to mark the answer wrong. In our DDD code, we should rather follow this rule, and try to make the implicit explicit.

### 24.4.3.4. Define your PODO classes

Main point is first to define your DDD Objects as plain Delphi class types - the famous PODOs, following the *Ubiquitous Language*. We will in fact define *Value Objects* class types, which may be grouped and nested to become *Entity Objects* or *Aggregates*.

To define a TPerson object, able to modelize a person identity, we may write the following classes:

```delphi
type
  /// Person full name
  TPersonFullName = class(TSynPersistent)
  protected
    fFirst: TFirstName;
    fMiddle: TMiddleName;
    fLast: TLastName;
  public
    function Equals(another: TPersonFullName): boolean; reintroduce;
    function FullName(country: TCountryIdentifier=ccUndefined): TFullName; virtual;
  published
    property First: TFirstName read fFirst write fFirst;
    property Middle: TMiddleName read fMiddle write fMiddle;
    property Last: TLastName read fLast write fLast;
  end;

  /// Person birth date
  TPersonBirthDate = class(TSynPersistent)
  protected
    fDate: TDateTime;
  public
    function Equals(another: TPersonBirthDate): boolean; reintroduce;
    function Age: integer; overload;
    function Age(FromDate: TDateTime): integer; overload;
  published
    property Date: TDateTime read fDate write fDate;
  end;

  /// Person object
  TPerson = class(TSynAutoCreateFields)
  protected
    fBirthDate: TPersonBirthDate;
    fName: TPersonFullName;
  public
    function Equals(another: TPerson): boolean; reintroduce;
  published
    property Name: TPersonFullName read fName;
    property Birth: TPersonBirthDate read fBirthDate;
  end;
```

First of all, you will see that we inherit from TSynPersistent and TSynAutoCreateFields. The benefit of those classes are the following:

- TSynPersistent has a virtual constructor, and a little less overhead than TPersistent, so may be preferred, especially when we will use *Dependency Injection and Interface Resolution* (page 417);
- TSynAutoCreateFields inherits from TSynPersistent, and its overridden Create will allocate all published class properties auto-magically - whereas its overridden Destroy will release those instances for you. As such, inheriting from TSynAutoCreateFields makes it a perfect fit for a Value Object, nesting sub objects as properties;
- Both have the RTTI enabled, so all published properties will be easily serialized as JSON (when used as DTO), or persisted later on on a database, when joined as Aggregate Roots.

In the above code, we defined TPerson.Name as a TPersonFullName class. So that we may use aPerson.Name.First or aPerson.Name.Last or even the runtime-computed aPerson.Name.FullName method which is able to display the full name, depending on per-country culture. We also reintroduced the Equals() method, which will allow to compare the objects per value, and not per reference.

Even if the birth date is just a date, we introduced a dedicated TPersonBirthDate class. The benefit is to have the overloaded Age() methods, which are pretty convenient in practice.

Once serialized as JSON, a TPerson content may be:

```json
{
   "Name": {
      "First": "John",
      "Middle": "",
      "Last": "Smith"
   },
   "Birth": {
      "Date": "1972-10-29"
   }
}
```

During the modelization phase, you will just define such class types, trying to reflect DDD's Ubiquitous Language into regular Delphi classes.

Take a look at the dddDomUserTypes.pas unit, to identify such patterns, and how we may be able to define an application user, gathering our TPerson class with a TAddress, in which a TCountry class will be used to store the corresponding country:

```delphi
/// a Person object, with some contact information
/// - an User is a person, in the context of an application
TPersonContactable = class(TPerson)
protected
   fAddress: TAddress;
   fPhone1: TPhoneNumber;
   fPhone2: TPhoneNumber;
   fEmail: TEmailAddress;
public
   function Equals(another: TPersonContactable): boolean; reintroduce;
published
   property Address: TAddress read fAddress;
   property Phone1: TPhoneNumber read fPhone1 write fPhone1;
   property Phone2: TPhoneNumber read fPhone2 write fPhone2;
   property Email: TEmailAddress read fEmail write fEmail;
end;
```

You can see that we did not pollute the class definition with any detail about persistence. What we did by now was to define a plain Value Object. We did not even specify that this class may be any Entity, nor introduce a primary key to identify it from a single access point. We found this way much cleaner that the approach of most other Java or C# DDD frameworks, which usually require to inherit from a parent Entity class, or use attributes to define the persistence expectations (like the primary key). We think that the domain types should not be polluted with those implementation details, and focus on expressing the model.
We will finally define a `TUser` **Entity** (or **Aggregate Root**), inheriting from `TPersonContactable`, i.e. modeling any application user account with all its personal information, with a flag to testify that its email was validated:

```pascal
TUser = class(TPersonContactable)
  private
    fLogonName: TLogonName;
    fEmailValidated: TDomUserEmailValidation;
  published
    property LogonName: TLogonName read fLogonName write fLogonName;
    property EmailValidated: TDomUserEmailValidation read fEmailValidated write fEmailValidated;
end;
```

Such a TPersistent-inheriting class could be used as a **Value Object** (or even a DTO), but become an **Entity** or **Aggregate** in the bounded context of the user account personal information. In order to store this data, we will now define an interface, implementing a **Persistence Service**.

### 24.4.3.5. Store your Entities in CQRS Repositories

When persisting our precious DDD Objects, the framework tries to follow some DDD patterns:

- Define **Aggregate Root** (or **Entities**) from **Value Objects**, as practical data context for storing the information;
- Use a **Repository** service to store those **Aggregates** instances;
- Follow CQRS (**Command Query Responsibility Segregation**) via a dedicated dual interface, splitting reads (**Queries**) and writes (**Commands**) in the **Repository** contract;
- Use **Factory** to instantiate CQRS Repository contracts on need.

In practice, we will use a **Factory** to create **Repository** class instances implementing the CQRS service methods, defined as a hierarchy of interface types, for a given **Aggregate Root**.

Let's start from an example, i.e. implement CQRS Repository services for our `TUser` class.

#### 24.4.3.5.1. CQRS Interfaces

The `mORMotDDD.pas` unit defines the following interface, which will benefit of being the root interface of all **Repository** services:

```pascal
type
  1CQRSService = interface(IInvokable)
    ['{923614C8-A639-45AD-A3A3-4548337923C9}']
    function GetLastError: TCQRSResult;
    function GetLastErrorInfo: variant;
  end;
```

This interface does nothing but allowing a generic access to the last error which occurred. This will be used instead of `Exception`, via the `TCQRSResult` enumeration, as a safe way of handling errors in a remote **Service**.

Exceptions are very convenient when running code in a process, but are difficult to handle over a remote connection, since the execution context is spread on both client and server sides. It is very difficult to propagate an exception raised on the server side to the client side, without leaking the server implementation. For instance, the SOAP standard provides a way of transmitting execution errors as dedicated XML messages - but it turns out to be a very verbose and complex path.

In `mORMot`, we defined a generic way of sending errors to the client side, for CQRS Services. By convention, any method will be defined as a function, returning its execution state as a `TCQRSResult` enumeration. If cqrssSuccess is returned, no error did happen on the server side, and execution may continue on the client side. Otherwise, an error "kind" is specified in the `TCQRSResult` transmitted value, and additional information is available as `string` or a `TDocVariant` custom **variant type**.
112) in the ICQRSService GetLastErrorInfo method. This allows to safely handle any kind of execution error on the client side, without the need to define dedicated exceptions. As we already stated about Error handling (page 458), exception should be exceptional - please refer to this paragraph for more details, including the benefit of that any stubbed or mocked interface will return cqrsSuccess (i.e. 0) by default, so let the test pass.

For our TUser CQRS Repository service, we will therefore define two interface types, one inheriting from ICQRSService for the Queries methods, and another one inheriting from this later interface to define the Commands methods:

CQRS Repository Service Interface for TUser

In dddDomUserCQRS.pas, we therefore defined two interface types, one IDomUserQuery for the read operations (i.e. Queries) of TUser aggregates, and an inherited IDomUserCommand for the write operations (i.e. Commands) of TUser aggregates.

We may argue that IDomUserCommand inheriting from IDomUserQuery is actually a violation of the Command Query Responsibility Segregation principle. Here, Commands are tied to Queries. Of course, we may have defined two diverse interfaces, both inheriting from ICQRSService as parent:

CQRS Dogmatic Repository Service Interface for TUser

Nothing prevent you from doing this. But in our case, especially with the mORMot underlying ORM, or a RDBMS database, the benefit is not obvious - sounds more like a dogmatic approach. To update a resource, you will need two interfaces: one IDomUserQuery instance to retrieve the existing value object, then one IDomUserCommand to modify it. From our pragmatic point of view, it is not mandatory. Also note that interface inheritance may differ from actual implementation class inheritance. IDomUserCommand may inherit from IDomUserQuery, but, e.g. if performance matters, you may still be able to implement a plain IDomUserQuery service with a dedicated class, on a separated database. In our case, interface inheritance is a common way of increasing code reuse. So if you want to be dogmatic about CQRS, you could - but only if it is worth the effort.

24.4.3.5.2. Queries Interface

Since we will separate queries and commands, we will first define the interface for actually reading TUser information:
As we stated previously, all those methods do return a TCQRSResult enumeration, which will be used on the service consumer side to notify on any execution error.

Instances of those interface will in fact have a limited life-time. To access the TUser persistence layer, a CQRS interface will be injected - via Dependency Injection and Interface Resolution (page 417), then allow to handle one or several TUser instances.

For queries, you could use IDomUserQuery.SelectByLogonName, IDomUserQuery.SelectByLastName or IDomUserQuery.SelectByEmailValidation methods to initialize a request. As you can see, there is no mention of primary key or ID in this interface definition. Even if under the hood, the implementation may use our ORM, and a TSQLRecord with its TSQLRecord.ID: TID property, the CQRS interface themselves make not those implementation details appear - unless it will be necessary. In our use case of an application targeting a single user, it is enough to be able to retrieve a user by its logon name, or by its last name.

If the Select* method executed without error (i.e. returned cqrsSuccess), we can later on retrieve the content by calling:
- IDomUserQuery.Get for filling the properties of a single already existing TUser object;
- IDomUserQuery.GetAll to return a list of TUser instances - for storage, we will use a TUserObjArray dynamic array, which should be released by the caller using ObjArrayClear() on the result variable;
- IDomUserQuery.GetNext to retrieve the actual matching TUser, one by one, following the principle of a database cursor;
- IDomUserQuery.GetCount will return the number of items matching the Select*.

Since the IDomUserQuery interface has a lifetime, you could call IDomUserQuery.Get or IDomUserQuery.GetAll several times after a single Select*. Note that in the common ORM-based implementation we will define below, the TUser information is actually retrieved and stored in memory by the Select* method.

Note that in the IDomUserQuery contract, the IDomUserQuery.HowManyValidatedEmail method, on the other hand, is stateless, and could be used without any prior Select*. Such methods may appear, depending on the Domain expectations.

The main point here is that, when defining your CQRS interface, you should focus on which data you need to access, in the most convenient way for you, and forget about the real persistence implementation - i.e. how data is stored. This is called, in DDD methods, as Persistence Ignorance, and is a very convenient way of uncoupling your business logic from actual technical details. If you was never asked by your commercials to support a new database engine, or even be able to switch from a SQL to a NoSQL storage, or an existing legacy proprietary obscure database used by a given customer... you are a lucky programmer, but - you know - it happens in real life!

Another advantage of starting from what you need in your domain, by using interface types as
contracts, is that you will probably focus on the domain, and may avoid the risk of an *anemic domain model* symptom, which appears when your persistence service is just a CRUD operation in disguise. If we need only CRUD operations, an ORM, or even plain SQL is enough. But if we want to have our domain code follow the ubiquitous language, and stick to the use cases of our business model, we should better design the persistence this way.

Last but not least, you will be able to *mock* or *stub* the persistence service - see *Stubs and mocks* (page 407), so ease unit test of your Domain code, without any dependency to any actual database layer. Following *Test Driven Design*, you will even be able to write the Domain core tests first, validate all your interfaces, even write the Application layer and test it with the current mock-up of the end-user application, and eventually finalize and tune the SQL or NoSQL storage at the final step, when the whole workflow is stabilized. It will help testing sooner, therefore fix sooner, and... hopefully release sooner.

### 24.4.3.5.3. Commands Interface

Following the CQRS (*Command Query Responsibility Segregation*) principle, we defined the write operations (i.e. *Commands*) in a separate interface. This type will inherit from `IDomUserQuery`, since it may be convenient to be able to first read the `TUser`, for instance before applying a modification to the stored information, like updating existing data, or adding some a missing entry.

```pascal
type
    IDomUserCommand = interface(IDomUserQuery)
        ['{D345854F-7337-4006-B324-5D635FBED312}']
        function Add(const aAggregate: TUser): TCQRSResult;
        function Update(const aUpdatedAggregate: TUser): TCQRSResult;
        function Delete: TCQRSResult;
        function DeleteAll: TCQRSResult;
        function Commit: TCQRSResult;
        function Rollback: TCQRSResult;
    end;
```

The main method of this *Command* interface is Commit. Following the *dual-phase commit* pattern, nothing will be written to the actual persistence storage unless this `IDomUserCommand.Commit` method is actually called.

In short, you query then update your data using the other Add/Update/Delete/... methods, then you run Commit.

For instance, to modification an existing record, you will call:
- `IDomUserQuery.SelectByLogonName`;
- `IDomUserCommand.Update`;

If the logon name is unknown, an error will raise at the first step. If the updated modification transmitted at the second step is invalid (i.e. you forgot to fill a mandatory field, or a value which should be unique, like a serial number, appear to exist already), then another error will be reported. But even after a successful Update, nothing will be stored in the database. Why? Because in most use cases, you will probably need to synchronize several operations: for instance, you may have to send an email, or call a third-party service, and write the new data only if everything was right. As such, you will need a two-phase write operation: first, you prepare and validate your data on each involved service, then, once everyone did give its green light, you eventually launch the process, which is, in the case of a persistence layer, calling Commit. In a real application, an unexpected low-level error may happen during the Commit phase - e.g. a network failure, a concurrency issue, or a problem between a chair and a keyboard - but it will not be likely to happen often. The *dual-phase* commit will ensure that
most errors will be identified during the first phase, using our ORM's Filtering and Validating (page 172) abilities.

Of course, if you want to run the IDomUserCommand.Add method, no prior IDomUserQuery.Select* call is mandatory. But for Update and Delete or DeleteAll commands, you will need first to define the data extend you will work on, by a previous call to Select*.

To use those CQRS interfaces, you could use IoC as usual:

```pascal
var cmd: IDomUserCommand;
    user: TUser;
    itext: RawUTF8;
...
aServer.Services.Resolve(IDomUserCommand,cmd);
user := TUser.Create;
try
    for i := 1 to MAX do begin
        UInt32ToUtf8(i,itext);
        user.LogonName := ' ' + itext;
        user.EmailValidated := evValidated;
        user.Name.Last := 'Last' + itext;
        user.Name.First := 'First' + itext;
        user.Address.Street1 := 'Street ' + itext;
        user.Phone1 := itext;
        if cmd.Add(user)<>cqrsSuccess then
            raise EMyApplicationException.CreateFmt('Invalid data: %s',[cmd.GetLastErrorInfo]);
    end;
    // here nothing is actually written to the database
    if cmd.Commit<>cqrsSuccess then
        raise EMyApplicationException.CreateFmt('Commit error: %s',[cmd.GetLastErrorInfo]);
    // here everything has been written to the database
finally
    user.Free;
end;
```

This dual-phase commit appears to be a clean way of implement the Unit Of Work pattern (page 353). Under the hood, when used with our ORM - as we will now explain - Unit Of Work will be expressed as a I*Command service, uncoupled from the persistence layer it runs on.

24.4.3.5.4. Automated Repository using the ORM

As you may have noticed, we did just defined the interface types we needed. That is, we have the contract of our persistence services, but no actual implementation of it. As such, those interface definitions are useless. Luckily for us, the mORMotDDD.pas unit offers an easy way to implement those using Object-Relational Mapping (page 130), with minimal coding.

24.4.3.5.4.1. DDD / ORM mapping

First we will need to map our domain object (i.e. our TUser instance and its properties) into a TSQLRecord. We may do it by hand, but you may find an handy way. Just run the following in the context of your application:

```pascal
TDDDRepositoryRestFactory.ComputeSQLRecord(TUser);
```

This class procedure will create a ddsqrcode.inc file in the executable folder, containing the needed field definition, with one TSQLRecord type corresponding to each hierarchy level of the original TPersistent definition. Nested fields will be defined as a single column in the TSQLRecord, e.g. Address.Country.Iso will be flattened as a Address_Country property.

So if we follow the class hierarchy, we will have:
CQRS Class Hierarchy Mapping for ORM and DDD Entities

Which will be defined as such in the ddsq1record.inc generated content:

```plaintext
type
  TSQLRecordPerson = class(TSQLRecord)
  ...
  published
    property Name_First: RawUTF8 read fFirst write fFirst;
    property Name_Middle: RawUTF8 read fMiddle write fMiddle;
    property Name_Last: RawUTF8 read fLast write fLast;
    property Birth: TDateTime read fBirthDate;
  end;

  TSQLRecordPersonContactable = class(TSQLRecordPerson)
  ...
  published
    property Address_Street1: RawUTF8 read fStreet1 write fStreet1;
    property Address_Street2: RawUTF8 read fStreet2 write fStreet2;
    property Address_CityArea: RawUTF8 read fCityArea write fCityArea;
    property Address_City: RawUTF8 read fCity write fCity;
    property Address_Region: RawUTF8 read fRegion write fRegion;
    property Address_Code: RawUTF8 read fCode write fCode;
    property Address_Country: integer read fCountry;
    property Phone1: RawUTF8 read fPhone1 write fPhone1;
    property Phone2: RawUTF8 read fPhone2 write fPhone2;
    property Email: RawUTF8 read fEmail write fEmail;
  end;

  TSQLRecordUser = class(TSQLRecordPersonContactable)
  ...
  published
    property LogonName: RawUTF8 read fLogonName write fLogonName;
    property EmailValidated: TDomUserEmailValidation read fEmailValidated write fEmailValidated;
  end;
```

In practice, the following property will need to be tuned as such:

```plaintext
property LogonName: RawUTF8 read fLogonName write fLogonName
  stored AS UNIQUE;
```

Take a look at the dddInfraRepoUser.pas and dddDomUserTypes.pas units to make a comparison between the DDD objects and their corresponding TSQLRecord* types.

You may wonder why we will introduce a separate level of classes, between the DDD Aggregates and the database engine. Why not directly persist the Domain objects (as most DDD implementations do)?

In fact, our approach has several benefits:
- Most of the time, simple mapping will be done automatically: once you called TDDDRepositoryRestFactory.ComputeSQLRecord, there is a very little additional coding to be done;
- But you still have access to the full mapping process, not using attributes (which may sound
convenient, but are polluting the DDD classes definition), but at Persistence Service method level;
- You could persist the same DDD classes as Value Objects, Entities or Aggregates, depending on the use context, by a custom mapping over a dedicated Persistence Service - your domain objects are uncoupled from their use context - remember that the same Value Object may become an Aggregate, or an Entity, depending on the context: why define again and again the same classes? just reuse the same tuned types via Composition (page 597);
- No need to inherit your DDD classes from a parent Entity class, or pollute it with an ID field (as most DDD implementations do);
- TSQLRecord allows to be truly persistent agnostic: you may do the storage on a regular RDBMS engine, on a NoSQL database, or in memory, at runtime, without touching your DDD objects;
- Practice did show that introducing ORM concepts at DDD classes level: just think about how the ID field may break your modelization, since the same object may be a Value Object in a context (so without any ID), but an Entity or an Aggregate in another context (so an ID is probably needed there) - it does indeed break the Persistence Ignorance pattern, and tend to produce an anemic domain model, i.e. CRUD operations in disguise;
- TSQLRecord classes give you direct access to how your data will be actually stored: most ORMs, when dealing with complex classes (like our Domain objects), tend to hide the mapping complexity, and therefore make it difficult to debug and tune the storage itself: which object field is mapped to which column? which tables are involved and joined for the queries? - whereas TSQLRecord make it clear how data will actually been stored: you may consider the TSQLRecord properties as a map of the SQL storage columns, or as the document stored in a NoSQL engine - database reuse and tuning will definitively be easier, when the TSQLRecord type definition shows you e.g. where the indexes should be created;
- You are not tied to use TSQLRecord: you can easily define a mORMotDDD.pas CQRS repository service fully abstracted from mORMot’s ORM, e.g. using existing tuned SQL statements, or any other mean of storage;
- Also consider that you are able to easily Stubs and mocks (page 407) the CQRS persistence service, whereas a direct ORM-oriented implementation will force you to create fake databases.

If you worry about performance of adding such a layer, you may be confident it won’t be a bottleneck: the CQRS mapping shares the same code than the framework ORM for RTTI and marshalling. Mapping process is just a fast loop over the properties, using cached RTTI, and as signing all content by reference, avoiding most memory allocations or content transformation.

24.4.3.5.4.2. Define the Factory

Since the generated TSQLRecordUser type follows known conventions, the mORMotDDD.pas unit is able to do almost all the persistence work in an automated way, by inheriting of two classes:
- Defining a Repository Factory (i.e. a class able to generate IDomUserQuery or IDomUserCommand instances on requests) by inheriting from TTDDDRepositoryRestFactory
- Defining the actual IDomUserCommand methods by inheriting from TTDDDRepositoryRestCommand, and using high level protected methods to access the TUser from internal TSQLRecordUser ORM values.

First of all, we define the Factory:

type
  TInfraRepoUserFactory = class(TDDDRepositoryRestFactory)
  public
    constructor Create(aRest: TSQLRest; aOwner: TTDDDRepositoryRestManager=nil); reintroduce;
  end;

constructor TInfraRepoUserFactory.Create(aRest: TSQLRest;
As you can see, the main point of this constructor is to supply the right parameters to the inherited TDDRRepositoryRestFactory.Create:

- We would like to implement a IDomUserCommand contract - and, by the way, implement also its parent IDomUserQuery interface;
- The actual implementation class will be TInfraRepoUser - which will be defined just after;
- The Aggregate/Entity class is a TUser kind of object;
- The associated TSQLRest server will be the one supplied to this class;
- The ORM class, defining the actual SQL table or NoSQL collection which will store the data, is TSQLRecordUser;
- An optional TDDDRepositoryRestManager instance may be supplied as owner of this factory - but it is not used in most cases.

The AddFilterOrValidate() method allows to set some Filtering and Validating (page 172) expectations at DDD level. Those rules will be applied before Commit will take place, without any use of the ORM rules. In the above code, TSynFilterTrim will remove any space from all text fields of the TUser instance, and TSynValidateNonVoidText will ensure that the TUser.LogonName field will not be ' ' - after space trimming. You may consider those rules as the SQL constraints you may be used to. But since they will be defined at DDD level, they will apply on any database back-end, even if it does not support any constraint - e.g. if it is a NoSQL engine, or a third-party persistence service you do not have the hand on.

You will probably want to use those CQRS interfaces, via usual IoC, at TSQLRest level, just like any Client-Server services via interfaces (page 419):

```pascal
var cmd: IDomUserCommand;
...
  aServer.Services.Resolve(IDomUserCommand, cmd);
```
or, for a Query:

```pascal
var qry: IDomUserQuery;
...
  aServer.Services.Resolve(IDomUserQuery, qry);
```

In order to be able to get a IDomUserCommand or IDomUserQuery instance from aServer.Services.Resolve(), you will need to register the TInfraRepoUserFactory first:

```pascal
aServer.ServiceContainer.InjectResolver([TInfraRepoUserFactory.Create(aServer)],true);
```
or if you want to maintain the factory instance life-time (e.g. to share it with other interface resolvers):

```pascal
var factory: TInfraRepoUserFactory;
...
  factory := TInfraRepoUserFactory.Create(aServer);
  try
    aServer.ServiceContainer.InjectResolver([factory]);
  ...
  finally
    factory.Free;
end;
```

This single TInfraRepoUserFactory will allow to implement both IDomUserCommand and IDomUserQuery contracts.

Of course, having the ability to let aServer own the factory, via the InjectResolver([...],true)
parameter, sounds easier to work with.

In practice, for a Client/Server environment, you may write:

```delphi
// Server side
RestServer := TSQLRestServerFullMemory.CreateWithOwnModel([TSQLRecordUser]);
...
RestServer.ServiceContainer.InjectResolver([TInfraRepoUserFactory.Create(RestServer)],true);
RestServer.ServiceDefine(TInfraRepoUser,[IDomUserCommand,IDomUserQuery],sicClientDriven);
// now you can use the services on the Server side
if RestServer.Services.Resolve(IDomUserCommand,cmd) then
  ... use cmd
if RestServer.Services.Resolve(IDomUserQuery,qry) then
  ... use qry
...
// Client side
RestClient := TSQLRestClientURIDll.Create(TSQLModel.Create(...),@URIRequest);
...
RestClient.ServiceDefine([IDomUserCommand],sicClientDriven);
// now you can use the services on the Client side
if RestServer.Services.Resolve(IDomUserCommand,cmd) then
  ... use cmd
if RestServer.Services.Resolve(IDomUserQuery,qry) then
  ... use qry
```

Note that InjectResolver() should be called **before** ServiceDefine(), otherwise the IoC won't take place as expected, and the TInfraRepoUserFactory class will be nil.

The CQRS services should be defined as **sicClientDriven** - and not as **sicSingle** or **sicShared**, since their lifetime is expected to be synchronized by the consumer side, i.e. the interface variable use on the client side.

On the client side, defining **IDomUserCommand** is enough to be able to use both **IDomUserCommand** and **IDomUserQuery** services, but on the server side you will have to explicitly define both interfaces, otherwise the Client/Server contracts won't match and you will not be able to use **IDomUserQuery** from the client side.

You could check the TInfraRepoUserFactory.RegressionTests method, as defined in dddInfraRepoUser.pas, to find out how such services may be defined and consumed.

### 24.4.3.5.4.3. Implement the CQRS methods

We have defined the factory, and registered the services. Now we define the needed methods of **IDomUserCommand** and **IDomUserQuery** in our custom class:

```
type
  TInfraRepoUser = class(TDDDRepositoryRestCommand,IDomUserCommand,IDomUserQuery)
  public
    function SelectByLogonName(const aLogonName: RawUTF8): TCQRSResult;
    function SelectByEmailValidation(aValidationState: TDomUserEmailValidation): TCQRSResult;
    function SelectByLastName(const aName: TLastName; aStartWith: boolean): TCQRSResult;
    function Get(out aAggregate: TUser): TCQRSResult;
    function GetAll(out aAggregates: TUserObjArray): TCQRSResult;
    function GetNext(out aAggregate: TUser): TCQRSResult;
    function Add(const aAggregate: TUser): TCQRSResult;
    function Update(const aUpdatedAggregate: TUser): TCQRSResult;
    function HowManyValidatedEmail: integer;
  end;
```

Note that we defined the TInfraRepoUser class as implementing both interface we need, via `= class(...,IDomUserCommand,IDomUserQuery)`. We need both types to be explicit in the class type definition, otherwise, IoC - i.e. `aServer.Services.Resolve()` calls - won't work for both.
As you can see, some methods appear to me missing. There is no Commit, nor Delete - which are required by IDomUserCommand. But in fact, those commands are so generic that they are already implemented for you in TDDDRepositoryRestCommand!

What we need know is to implement those methods, using the internal protected ORM*() methods inherited by this parent class:

```pascal
function TInfraRepoUser.SelectByLogonName(const aLogonName: RawUTF8): TCQRSResult;
begin
  result := ORMSelectOne('LogonName=?',[aLogonName],(aLogonName=''));
end;

function TInfraRepoUser.SelectByEmailValidation(aValidationState: TDomUserEmailValidation): TCQRSResult;
begin
  result := ORMSelectAll('EmailValidated=?',[ord(aValidationState)]);
end;

function TInfraRepoUser.SelectByLastName(const aName: TLastName; aStartWith: boolean): TCQRSResult;
begin
  if aStartWith then
    result := ORMSelectAll('Name_Last LIKE ?',[aName+'%'],(aName=''))
  else
    result := ORMSelectAll('Name_Last=?',[aName],(aName=''));
end;

function TInfraRepoUser.Get(out aAggregate: TUser): TCQRSResult;
begin
  result := ORMGetAggregate(aAggregate);
end;

function TInfraRepoUser.GetAll(out aAggregates: TUserObjArray): TCQRSResult;
begin
  result := ORMGetAllAggregates(aAggregates);
end;

function TInfraRepoUser.GetNext(out aAggregate: TUser): TCQRSResult;
begin
  result := ORMGetNextAggregate(aAggregate);
end;

function TInfraRepoUser.Add(const aAggregate: TUser): TCQRSResult;
begin
  result := ORMAdd(aAggregate);
end;

function TInfraRepoUser.Update(const aUpdatedAggregate: TUser): TCQRSResult;
begin
  result := ORMUpdate(aUpdatedAggregate);
end;

function TInfraRepoUser.HowManyValidatedEmail: integer;
begin
  if ORMSelectCount('EmailValidated=%',[ord(evValidated)],[],result)<>cqrsSuccess then
    result := 0;
end;
```

Almost everything is already defined at TDDDRepositoryRestCommand level. Our TInfraRepoUser class, implementing a full CQRS service, fully abstracted from the ORM, is implemented by a few internal ORM*() method calls.

All the error handling, including server-side exception catching, and conversion into TCQRSResult / ICQRSService.GetLastErrorInfo content, is already implemented in
TDDDRepositoryRestCommand.

All the data access via the TSQLRecordUser REST persistence layer, with any Filtering and Validating (page 172) defined rule, is also incorporated in TDDDRepositoryRestCommand. The conversion to/from TUser properties has been optimized, so that fields will be moved by reference, with no memory allocation nor content modification, for best performance and data safety. The type mapping specified by TInfraRepoUserFactory.Create is enough to make the whole process as automated as possible.

In fact, our TInfraRepoUser class is just a thin wrapper forcing use of strong typing in its methods parameters (i.e. using TUser/TUserObjArray whereas the ORM*() methods are more relaxed about actual typing), and ensuring that the ORM specificities are followed as expected, e.g. a search against the TUser.Name.Last field will use the TSQLRecordUser.Name_Last ORM column, with the proper LIKE operator.

Internally, TDDDRepositoryRestCommand.ORMPrepareForCommit will call all DDD and ORM TSynFilter and TSynValidate rules, as previously defined. It sounds indeed like a real advantage not to wait until the database layer is reached, to have those constraints verified. The sooner an error is notified, the better - especially in a complex SOA system.

Under the hood, TDDDRepositoryRestCommand will define a TSqlRestBatch - see BATCH sequences for adding/updating/deleting records (page 350) - for storing all write commands in memory (as JSON) - e.g. cmd.Add - and will send them to the database engine, with optimized SQL or NoSQL statements, only when cmd.Commit will be executed.

24.4.3.6. Isolate your Domain using DTOs

DDD's DTO may also be defined as record, and directly serialized as JSON via text-based serialization. Don't be afraid of writing some translation layers between TSQLRecord and DTO records or, more generally, between your Application layer and your Presentation layer. It will be very fast, on the server side. If your service interfaces are cleaner, do not hesitate.

But defining DTO types, just for uncoupling, may become time consuming. If you start writing a lot of wrapping code, forget about it, and expose your Domain Value Objects or even your Entities, as stated above. Or automate the wrapper coding, using RTTI and code generators. You have to weight the PROs and the CONs, like always... And never forget to write proper unit testing of this marshalling code, since it may induce some unexpected issues.

If you expect your DDD's objects to be schema-less or with an evolving structure (e.g. for DTO), depending on each context, you may benefit of not using a fixed type like class or record, but use TDocVariant custom variant type (page 112). This kind of variant will be serialized as JSON, and allow late-binding access to its properties (for object documents) or items (for array documents). In the context of interface-based services, using per-reference option at creation (i.e. _ObjFast() _ArrFast() _JsonFast() _JsonFmtFast() functions) does make sense, in order to spare the server resources.

24.4.4. Defining services

In practice, mORMot's Client-Server architecture may be used as such:
- Services via methods - see Client-Server services via methods (page 373) - can be used to publish methods corresponding to your aggregate roots defined as TSQLRecord. This will make it pretty RESTful compatible.
- Services via interfaces - see Client-Server services via interfaces (page 419) - can be used to publish
all your processes.
Dedicated factories can be used on both Client and Server side, to define your repositories and/or domain operations.

Client-Server services via methods (page 373) may be preferred if you expect your service to be consumed in a truly RESTful way. But since in DDD you should better protect your Domain via a dedicated Adapter layer, such compatibility should be an implementation smell. In practice, Client-Server services via interfaces (page 419) will offer better integration and automation of its process, e.g. parameter type validation (with JSON marshalling), session handling, interface-level multi-threading and security abilities, logging, ability to be emulated via Stubs and mocks (page 407), and - last but not least - Publish-subscribe for events (page 449).

24.4.5. Event-Driven Design

Event-Driven could be implemented in mORMot by at least two ways:
- Using interface callbacks of the framework Client-Server services via interfaces (page 419);
- Storing the system states in a table, and let Real-time synchronization (page 183) generate the events.

Both ways have their own benefits and drawbacks, and you may pick up the one which match your particular use case. The first may be more easy to implement and versatile to use, but the second will work with off-line periods, and

24.4.5.1. Events as Callbacks

DDD's Events could easily be implemented as Asynchronous callbacks (page 444), when an interface callback is defined as Service Methods Parameters (page 423). In this case, the interface type will define the various DDD events, ready to be notified and propagated in real-time across the whole system.

An application layer may provide a specific callback to the domain, which will push the notification as a regular Delphi call, but in fact transmitted via WebSockets from the corresponding Domain Service to the right application layer. The current implementation relies on WebSockets for remote access, but other protocols may be available in the future, since interface parameters callbacks may be implemented by any actual transmission class.

No need to encapsulate your events within a dedicated message class (as most Event-Driven implementations require), or pollute your Domain code to follow a fixed protocol expectations: just run a notification method corresponding to the event, and you are done - all subscribers will be notified.

No need to put in production a Message bus, or a centralized system. Using callbacks, you will your outer layers (e.g. Application or Presentation layers) be cleanly notified by the Domain Services, without any waste of resource, and without potential bottleneck. Each node of your system will communicate directly with its subscriber, from a pure interface method call, as if it was a local process. See Publish-subscribe for events (page 449) for implementation details.

In practice, the callbacks may be propagated from the Domain layer to the Application or Presentation layers, which may also have their own callbacks definitions, using not Domain objects, but their own DTOs. Marshalling an event will be as easy as writing a class implementing an I*Callback interface as defined in the Domain, translating its parameters into the DTO types are defined for the outer Application or Presentation services.
On the server side, you may even define the callbacks in the very same process, without the WebSockets overhead, but calling directly the Domain services, whose interface type will be defined at Domain level, but the class type implemented at Infrastructure level:

- The application layer may be able to run directly the Domain code in its own service/daemon, calling the actual implementation at Infrastructure level, with a single straight WebSockets transmission - the DDD's Adapters types being pure Delphi classes, running in process, with no overhead.
- Or, you may gather Domain Services in some specific stand-alone daemons, which may be able to cache the events, and/or centralize some process - as a benefit, it may help those services be truly stateless, so the Application Layer may become redundant for better scaling.

Using interface values and call their methods is a natural way of writing callbacks in Delphi code, very close to the VCL/RAD Events you may be used to, but with the benefit of the abstraction of Interfaces (page 385), especially SOLID design principles (page 389). If your need is to react in real time to some change of the system, they are probably the preferred way.

### 24.4.5.2. Event Sourcing via Event Oriented Databases

Another new, and popular DDD's Events implementation pattern is to define some kind of event persistence, which will be used as Event Sourcing. Here, we won't rely on explicit messages to transmit the events (as we just proposed via asynchronous interface callbacks), but we will use some state storage in an Event Oriented Persistence, then let subscribers by notified for each state change.

In this pattern, there are not directly any kind of Event defined. The state of the Domain is stored somewhere, then any change of state should be notified to whom has interest for it. Obviously, one potential easy implementation may be via Real-time synchronization (page 183), as proposed by the framework.

The Domain services - see e.g. Services (page 601) or if you Store your Entities in CQRS Repositories (page 610) - may modify a dedicated TSQLRecord table, which will contain only a small part of the state of the model. For instance, its TSQLRecord fields definition (page 131) may store only a few evolving values, like the latest order placed, or the price of an item, or the connection state of a peripheral. The main point is to restrict the data stored to its minimum, e.g. this evolving value and the name (or ID) of the object.

Thanks to the framework Real-time synchronization (page 183), any client process or service, via its own Slave copy of this TSQLRecord State storage, will be notified asynchronously. This notification will reflect each change of state, and will let the consumer react as expected. One OnNotify event is available, to track each individual change of state, as specified as parameter to TSQLRestServer.RecordVersionSynchronizeSlaveStart.

When using Events as Callbacks (page 621), you may miss some events: if the consumer service is off-line, there won't be any event notified. It may be as expected, but may be a huge issue in some cases. On the other hand, the Event Oriented Persistence model will allow the consumers to be safely off-line at some time. Each ORM Slave will have its own copy of the data, then will be able to retrieve all the missed changes of state, when it goes on-line.

This implementation pattern is in fact the base of any true Event Sourcing process. Following this DDD pattern, each node of the system should store the data it needs. The system nodes won't ask for a given information (e.g. "What is the current temperature?"), but will be notified of each temperature change, then store the value, and being able to propagate any incoming events with almost no dependency. The main benefit is that you could add some node to the system, without any prior knowledge of what is already there. Such Events-driven Architecture (EDA) or Domain Event Driven...
Service Oriented Architecture (D-EDA) may be complex to maintain and debug, once they reach a given size. For instance, some unexpected Event Cascade may happen, when you get a sequence of events triggering other events: you may induce an infinite rebound in the whole system. As a consequence, a "pure Event Driven" system will probably be a wrong idea. Event Sourcing may be introduced for some part of your Domain, where it does make better sense. See http://martinfowler.com/eaaDev/EventCollaboration.html, for more material.

As a side benefit, scaling of the whole system may be increased by this pattern. Each Event State storage may be seen as a safe cache of the system state, in the bounded context of a given set of values. When your business logic wonder about this particular state, it may ask this dedicated service, leveraging the main database. You may even consider storing the whole state history in a dedicated Audit Trail for change tracking (page 178) storage, without impacting the whole system.

24.4.6. Building a Clean architecture

A common DDD architecture is expressed as in the following model, which may look like a regular Multi-tier architecture (page 88) design at first, but should be implemented as a Clean Uncoupled Domain-Oriented Architecture (page 603):

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presentation</td>
<td>MVC UI generation and reporting</td>
</tr>
<tr>
<td>Application</td>
<td>Services and high-level adapters</td>
</tr>
<tr>
<td>Domain Model</td>
<td>Where business logic remains</td>
</tr>
<tr>
<td>Data persistence</td>
<td>ORM and external services</td>
</tr>
<tr>
<td>Cross-Cutting</td>
<td>Horizontal aspects shared by other layers</td>
</tr>
</tbody>
</table>

Physically, it involves a common n-Tier representation splitting the classical Logic Tier into two layers, i.e. Application layer and Domain Model layer. At logical level, DDD will try to uncouple the Domain Model layer from other layers, so the code itself will rely on interfaces and dependency injection to let the core Domain focus on the business logic, not on implementation details (e.g. persistence or communication).

The RESTful SOA components of our Synapse mORMot framework can therefore define such an Architecture:
Clean Domain-Oriented Architecture of mORMot

As we already stated, the main point of this Clean Architecture is to control coupling, and isolate the Domain core from the outer layers. In Delphi, unit dependencies (as displayed e.g. by our SynProject tool) will be a good testimony of proper objects uncoupling: in the units defining your domain, you may split it between Domain Model and Domain Services (the 2nd using the first, and not vice-versa), and you should never have any dependency to a particular DB unit, just to the framework's core units, i.e. SynCommons.pas and mORMot.pas. Interfaces in practice: dependency injection, stubs and mocks (page 406) - via Client-Server services via interfaces (page 419) or at ORM initialization level - will ensure that your code is uncoupled from any low-level technical dependency. It will also allow proper testing of your application workflows, e.g. stubbing the database if necessary.

In fact, since a Service-Oriented Architecture (SOA) (page 90) tends to ensure that services comprise unassociated, loosely coupled units of functionality that have no calls to each other embedded in them, we may define two levels of services, implemented by two interface factories, using their own hosting and communication:
- One set of services at Application layer, to define the uncoupled contracts available from Client applications;
- One set of services at Domain Model layer, which will allow all involved domains to communicate with each other, without exposing it to the remote clients.
Therefore, those layers could be also implemented as such:

![Diagram of Domain-Oriented Architecture of mORMot]

**Alternate Domain-Oriented Architecture of mORMot**

In order to provide the better scaling of the server side, cache can be easily implemented at every level, and hosting can be tuned in order to provide the best response time possible: one central server, several dedicated servers for application, domain and persistence layers...

Due to the SOLID design of mORMot - see SOLID design principles (page 389) - you can use as many Client-Server services layers as needed in the same architecture (i.e. a Server can be a Client of other processes), in order to fit your project needs, and let it evolve from the simplest architecture to a full scalable Domain-Driven design.
25. Testing and logging

Adopt a mORMot

Since we covered most architectural and technical aspects of the framework, it is time to put the last missing bricks to the building, meaning testing and logging.
25.1. Automated testing

You know that testing is (almost) everything if you want to avoid regression problems in your application.

How can you be confident that any change made to your software code won't create any error in other part of the software?

Automated unit testing is a good candidate for avoiding any serious regression.

And even better, testing-driven coding can be encouraged:
- Write a void implementation of a feature, that is code the interface with no implementation;
- Write a test code;
- Launch the test - it must fail;
- Implement the feature;
- Launch the test - it must pass;
- Add some features, and repeat all previous tests every time you add a new feature.

It could sounds like a waste of time, but such coding improve your code quality a lot, and, at least, it help you write and optimize every implementation feature.

The framework has been implemented using this approach, and provide all the tools to write tests. In addition to what other Delphi frameworks offer (e.g. DUnit / DUnitX), the SynTests.pas unit is very much integrated with other elements of the framework (like logging), is cross-platform and cross-compiler, and provides a complete stubbing / mocking mechanism to cover Interfaces in practice: dependency injection, stubs and mocks (page 406).

25.1.1. Involved classes in Unitary testing

The SynTests.pas unit defines two classes (both inheriting from TSynTest), implementing a complete Unitary testing mechanism similar to DUnit, with less code overhead, and direct interface with the framework units and requirements (UTF-8 ready, code compilation from Delphi 6 up to Delphi 10.3 Rio and FPC, no external dependency).

The following diagram defines this class hierarchy:

\[
\begin{aligned}
&\text{TSynTests} \\
&\quad \downarrow \\
&\text{TSynTest} \\
&\quad \downarrow \\
&\text{TSynTestCase}
\end{aligned}
\]

\[TSynTest\ classes\ hierarchy\]

The main usable class types are:
- TSynTestCase, which is a class implementing a test case: individual tests are written in the published methods of this class;
- TSynTests, which is used to run a suit of test cases, as defined with the previous class.

In order to define tests, some TSynTestCase children must be defined, and will be launched by a TSynTests instance to perform all the tests. A text report is created on the current console, providing statistics and Pass/Fail.
25.1.2. First steps in testing

Here are the functions we want to test:

```pascal
function Add(A,B: double): Double; overload;
begin
  result := A+B;
end;

function Add(A,B: integer): integer; overload;
begin
  result := A+B;
end;

function Multiply(A,B: double): Double; overload;
begin
  result := A*B;
end;

function Multiply(A,B: integer): integer; overload;
begin
  result := A*B;
end;
```

So we create three classes one for the whole test suit, one for testing addition, one for testing multiplication:

```pascal
type
  TTestNumbersAdding = class(TSynTestCase)
  published
    procedure TestIntegerAdd;
    procedure TestDoubleAdd;
  end;

TTestNumbersMultiplying = class(TSynTestCase)
  published
    procedure TestIntegerMultiply;
    procedure TestDoubleMultiply;
  end;

TTestSuit = class(TSynTests)
  published
    procedure MyTestSuit;
  end;
```

The trick is to create published methods, each containing some tests to process.

Here is how one of these test methods are implemented (I let you guess the others):

```pascal
procedure TTestNumbersAdding.TestDoubleAdd;
var A,B: double;
  i: integer;
begin
  for i := 1 to 1000 do
  begin
    A := Random;
    B := Random;
    CheckSame(A+B,Adding(A,B));
  end
end;
```

The `CheckSame()` is necessary because of floating-point precision problem, we can't trust plain = operator (i.e. `Check(A+B=Adding(A,B))` will fail because of rounding problems).

And here is the test case implementation:
procedure TTestSuit.MyTestSuit;
begin
  AddCase([TTestNumbersAdding,TTestNumbersMultiplying]);
end;

And the main program (this .dpr is expected to be available as a console program):

    with TTestSuit.Create do
    try
      ToConsole := @Output; // so we will see something on screen
      Run;
      readln;
    finally
      Free;
    end;

Just run this program, and you'll get:

    Suit
    ------

1. My test suit

   1.1. Numbers adding:
       - Test integer add: 1,000 assertions passed 92us
       - Test double add: 1,000 assertions passed 125us
       Total failed: 0 / 2,000 - Numbers adding PASSED 360us

   1.2. Numbers multiplying:
       - Test integer multiply: 1,000 assertions passed 73us
       - Test double multiply: 1,000 assertions passed 117us
       Total failed: 0 / 2,000 - Numbers multiplying PASSED 324us

Generated with: Delphi 7 compiler

Time elapsed for all tests: 1.51ms
Tests performed at 25/03/2014 10:59:33

Total assertions failed for all test suits: 0 / 4,000
All tests passed successfully.

You can see that all text on screen was created by "UnCamelCasing" the method names (thanks to our good old Camel), and that the test suit just follows the order defined when registering the classes. Each method has its own timing, which is pretty convenient to track performance regressions.

This test program has been uploaded in the SQLite3\Sample\07 - SynTest folder of the Source Code Repository.

25.1.3. Framework test coverage

The SAD # DI-2.2.2 (page 2549) defines all classes released with the framework source code, which covers all core aspects of the framework. Global testing coverage is good, excellent for core components (more than 25,000,000 individual checks are performed for revision 1.18), but there is still some User-Interface related tests to be written.

Before any release all unitary regression tests are performed with the following compilers:

- Delphi 5 (for a limited scope, including SynCommons.pas, SynLog.pas, SynTests.pas, SynCrypto.pas, SynEcc.pas, SynPdf.pas and SynDB.pas);
- Delphi 6;
- Delphi 7, with and without our Enhanced Run Time Library;
- Delphi 2007;
- Delphi 2010 (we assume that if it works with Delphi 2010, it will work with Delphi 2009, with the exception of generic compilation);
- Delphi XE4;
- Delphi XE6;
- Delphi XE7;
- Delphi 10 Seattle;
- Delphi 10.1 Berlin;
- Delphi 10.2 Tokyo;
- Delphi 10.3 Rio;
- CrossKylix 3.0;
- FPC 3.x - preferred is 3.2 fixes.

Target platforms are Win32 and Win64 for Delphi and FPC, plus Linux 32/64 for FPC and CrossKylix.

Then all sample source code (including the Main Demo and SynDBExplorer sophisticated tools) are compiled, and user-level testing is performed against those applications.

You can find in the compil.bat and compilpil.bat files of our source code repository how incremental builds and tests are performed.
25.2. Enhanced logging

A logging mechanism is integrated with cross-cutting features of the framework. It includes stack trace exception and such, just like *MadExcept*, using `.map` file content to retrieve debugging information from the source code.

Here are some of its features:

- Logging with a set of levels, not only a level scale;
- Fast, low execution overhead;
- Can load `.map` file symbols to be displayed in logging (i.e. source code file name and line numbers are logged instead of a hexadecimal value);
- Compression of `.map` into binary `.mab` (900 KB -> 70 KB);
- Inclusion of the `.map`/`.mab` into the `.exe`, with very slow size increase;
- Exception logging (*Delphi* or low-level exceptions) with unit names and line numbers;
- Optional stack trace with units and line numbers;
- Methods or procedure recursive tracing, with *Enter* and *auto-Leave* (using a fake interface instance);
- High resolution time stamps, for customer-side profiling of the application execution;
- Set/ enumerates / `TList` / `TPersistent` / `TObjectList` / dynamic array JSON serialization;
- Per-thread or global logging;
- Optional multiple log files on the same process;
- Optional rotation when main log reaches a specified size, with compression of the rotated logs;
- Integrated log archival (in `.zip` or any other format, including our `.synlz`);
- Optional colored echo to a console window, for interactive debugging;
- Fast log viewer tool available, including thread filtering and customer-side execution profiling;
- Optional remote logging via HTTP - the log viewer can be used as server;
- Optional events transmission to a UDP syslog server.

25.2.1. Setup logging

Logging is defined mainly by a per-class approach. You usually define your logging expectations by using a `TSynLog` class, and setting its `Family` property. Note that it is perfectly feasible to use your own `TSynLog` class instance, with its own `TSynLog` family settings, injected at the constructor level; but in *mORMot*, we usually use the per-class approach, via `TSynLog`, `TSQLLog`, `SynDBLog` and `SQLite3Log` - see below (page 641).

For sample code (and the associated log viewer tool), see "11 - Exception logging" folder in "Sqlite3\Samples".

In short, you can add logging to your program, just by using the `TSynLog` class, as such:

```delphi
TSynLog.Add.Log(sllInfo, Stats.DebugMessage);
```

This line will log the `Stats.DebugMessage` text, with a `sllInfo` notification level. See the description of all Log() overloaded methods of the `ISynLog` interface, to find out how your project can easily log events.

First of all, you need to define your logging setup via code:

```delphi
with TSynLog.Family do begin
  Level := LOG_VERBOSE;
  //Level := [sllException, sllExceptionOS];
  //HighResolutionTimestamp := true;
  //AutoFlushTimeOut := 5;
end;
```
OnArchive := EventArchiveSynLZ;
// OnArchive := EventArchiveZip;
ArchiveAfterDays := 1; // archive after one day
end;

The main setting here is TSynLog.Family.Level := ... which defines which levels are to be logged. That is, if sllInfo is part of TSynLog.Family.Level, any TSynLog.Add.Log(sllInfo,...) command will log the corresponding content - otherwise, it will be a no-operation. LOG_VERBOSE is a constant setting all levels at once.

You have several debugging levels available, and even 4 custom types:

```
TSynLogInfo = (sllNone, sllInfo, sllDebug, sllTrace, sllWarning, sllError,
    sllEnter, sllLeave,
    sllLastError, sllException, sllExceptionOS, sllMemory, sllStackTrace,
    sllFail, sllSQL, sllCache, sllResult, sllDB, sllHTTP, sllClient, sllServer,
    sllServiceCall, sllServiceReturn, sllUserAuth,
    sllCustom1, sllCustom2, sllCustom3, sllCustom4, sllNewRun);
```

Here are the purpose of each logging level:
- sllInfo will log general information events;
- sllDebug will log detailed debugging information;
- sllTrace will log low-level step by step debugging information;
- sllWarning will log unexpected values (not an error);
- sllError will log errors;
- sllEnter will log every method start;
- sllLeave will log every method quit;
- sllLastError will log the GetLastError OS message;
- sllException will log all exception raised - available since Windows XP;
- sllExceptionOS will log all OS low-level exceptions (EDivByZero, ERangeError,
  EAccessViolation...);
- sllMemory will log memory statistics;
- sllStackTrace will log caller's stack trace (it is by default part of TSynLogFamily.
  LevelStackTrace like sllError, sllException, sllExceptionOS, sllLastError and sllFail);
- sllFail was defined for TSynTestsLogged. Failed method, and can be used to log some
  customer-side assertions (may be notifications, not errors);
- sllSQL is dedicated to trace the SQL statements;
- sllCache should be used to trace any internal caching mechanism (it is used for instance by our
  SQL statement caching);
- sllResult could trace the SQL results, JSON encoded;
- sllDB is dedicated to trace low-level database engine features;
- sllHTTP could be used to trace HTTP process;
- sllClient/sllServer could be used to trace some Client or Server process;
- sllServiceCall/sllServiceReturn to trace some remote service or library;
- sllUserAuth to trace user authentication (e.g. for individual requests);
- sllCustom1..sllCustom4 items can be used for any purpose by your programs;
- sllNewRun will be written when a process re-opens a rotated log.

Logging is not using directly a TSynLogInfo level, but the following set:

```
/// used to define a logging level
// - i.e. a combination of none or several logging event
// - e.g. use LOG_VERBOSE constant to log all events
TSynLogInfos = set of TSynLogInfo;
```

Most logging tools in the wild use a level scale, i.e. with a hierarchy, excluding the lower levels when
Our logging classes use a set, and not directly a particular level, so you are able to select which exact events are worth recording. In practice, we found this pattern to make a lot of sense and to be much more efficient for support.

### 25.2.2. Call trace

The logging mechanism can be used to trace recursive calls. It can use an interface-based mechanism to log when you enter and leave any method:

```pascal
procedure TMyDB.SQLExecute(const SQL: RawUTF8);
var ILog: ISynLog;
begin
  ILog := TSynLogDB.Enter(self, 'SQLExecute');
  // do some stuff
  ILog.Log(sllInfo, 'SQL=%', [SQL]);
end;
// when you leave the method, it will write the corresponding event to the log
```

It will be logged as such:

```
20110325 19325801 +    MyDBUnit.TMyDB(004E11F4).SQLExecute
20110325 19325801 info   SQL=SELECT * FROM Table;
```

Note that by default you have human-readable time and date written to the log, but it is also possible to replace this timing with high-resolution timestamps. With this, you'll be able to profile your application with data coming from the customer side, on its real computer. Via the Enter method (and its auto-Leave feature), you have all information needed for this.

### 25.2.3. Including symbol definitions

In the above logging content, the method name is set in the code (as 'SQLExecute'). But if the logger class is able to find a .map file associated to the .exe, the logging mechanism is able to read this symbol information, and write the exact line number of the event.

By default, the .map file information is not generated by the compiler. To force its creation, you must ensure the {$D+} compiler directive is set in every unit (which is the case by default, unless you set {$D-} in the source), and the "Detailed Map File" option selected in the Project > Options > Linker page of the Delphi IDE.

In the following log entries, you'll see both high-resolution time stamp, and the entering and leaving of a TTestCompression.TestLog method traced with no additional code (with accurate line numbers, extracted from the .map content):

```
0000000000008056 +    TTestCompression(00A83570).000E6C79 SynSelfTests.TTestCompression.TestLog(376)
000000000001785 -
```

There is already a dedicated TSynLogFile class able to read the .log file, and recognize its content.

The first time the .map file is read, a .mab file is created, and will contain all symbol information needed. You can send the .map file with the .exe to your client, or even embed its content to the .exe (see the Map2Mab.dpr sample file located in the Samples\11 - Exception logging\ folder).

This .mab file is very optimized: for instance, a .map of 927,984 bytes compresses into a 71,943 .mab file.

### 25.2.4. Exception handling
Of course, this logging mechanism is able to intercept the raise of exceptions, including the worse (e.g. EAccessViolation), to be logged automatically in the log file, as such:

```plaintext
000000000000090B EXCOS EAccessViolation (C0000005) at 000E9C7A SynSelfTests.Proc1 (785) stack trace
```

The TSynLogInfo logging level makes a difference between high-level Delphi exceptions (sllException) and lowest-level OS exceptions (sllExceptionOS) like EAccessViolation.

For instance, if you add to your program:

```plaintext
uses
  SynLog;
(...)
TSynLog.Family.Level := [sllExceptionOS];
```

all OS exceptions (excluding pure Delphi exception like EConvertError and such) will be logged to a separated log file.

```plaintext
TSynLog.Family.Level := [sllException,sllExceptionOS];
```

will trace also Delphi exceptions, for instance.

You can specify some Exception class to be ignored, by adding them to Family.ExceptionIgnore internal list. It could make sense to add this setting, if your code often triggers some non-breaking exceptions, e.g. with StrToInt():

```plaintext
TSynLog.Family.ExceptionIgnore.Add(EConvertError);
```

If your Delphi code executes some .Net managed code (e.g. exposed via some COM wrapper components), the unit is able to recognize most unhandled .Net exceptions, and log them with their original C# class name (for instance, E01eSysError 80004003 will be recorded as a much more user-friendly " [.NET/CLR unhandled ArgumentNullException] " message.

You can set the following global variable to assign a customized callback, and be able to customize the logging content associated to any exception:

```plaintext
type
  /// global hook callback to customize exceptions logged by TSynLog
  /// - should return FALSE if Context.EAddr and Stack trace is to be appended
  TSynLogExceptionToStr = function(WR: TTextWriter; const Context: TSynLogExceptionContext): boolean;

var
  /// allow to customize the Exception Logging message
  TSynLogExceptionToStrCustom: TSynLogExceptionToStr = nil;
```

The Context: TSynLogExceptionContext content is to be used to append some text to the specified TTextWriter instance.

An easier possibility is to inherit your custom exception class from ESynException, and override its unique virtual method:

```plaintext
/// generic parent class of all custom Exception types of this unit
ESynException = class(Exception)
  /// can be used to customize how the exception is logged
  /// - this default implementation will call the DefaultSynLogExceptionToStr()
  /// - callback or TSynLogExceptionToStrCustom, if defined
  /// - override this method to provide a custom logging content
  /// - should return TRUE if Context.EAddr and Stack trace is not to be
  ///   written (i.e. as for any TSynLogExceptionToStr callback)
```
function CustomLog(WR: TTextWriter; const Context: TSynLogExceptionContext): boolean; virtual;
end;

See TSynLogExceptionContext to check the execution context, and the implementation of the function DefaultSynLogExceptionToStr() function.

### 25.2.5. Serialization

dynamic arrays can also be serialized as JSON in the log on request, via the default TSynLog class, as defined in SynLog.pas unit - see T DynArray dynamic array wrapper (page 107).

The TSQLLog class (using the enhanced RTTI methods defined in mORMot.pas unit) is even able to serialize TSQLRecord, TPersistent, TList and TCollection instances as JSON, or any other class instance, after call to TJSONSerializer. RegisterCustomSerializer.

For instance, the following code:

```pascal
procedure TestPeopleProc;
var People: TSQLRecordPeople;
    Log: ISynLog;
begin
    Log := TSQLLog.Enter;
    People := TSQLRecordPeople.Create;
    try
        People.ID := 16;
        People.FirstName := 'Louis';
        People.LastName := 'Croivébaton';
        People.YearOfBirth := 1754;
        People.YearOfDeath := 1793;
        Log.Log(sllInfo, People);
    finally
        People.Free;
    end;
end;
```

will result in the following log content:

```
0000000000001172  +    000E9F67 SynSelfTests.TestPeopleProc (784)
0000000000001718 info
{"TSQLRecordPeople(00AB92E0)":{"ID":16,"FirstName":"Louis","LastName":"Croivébaton","Data":"","YearOfBirth":1754,"YearOfDeath":1793}%
0000000000001731
```

### 25.2.6. Multi-threaded applications

You can define several log files per process, and even a per-thread log file, if needed (it could be sometimes handy, for instance on a server running the same logic in parallel in several threads).

The logging settings are made at the logging class level. Each logging class (inheriting from TSynLog) has its own TSynLogFamily instance, which is to be used to customize the logging class level. Then you can have several instances of the individual TSynLog classes, each class sharing the settings of the TSynLogFamily.

You can therefore initialize the "family" settings before using logging, like in this code which will force to log all levels (LOG_VERBOSE), and create a per-thread log file, and write the .log content not in the .exe folder, but in a custom directory:

```pascal
with TSynLogDB.Family do
begin
    Level := LOG_VERBOSE;
    PerThreadLog := ptOneFilePerThread;
    DestinationPath := 'C:\Logs';
end;
```
If you specify PerThreadLog := ptIdentifiedInOnFile for the family, a new column will be added for each log row, with the corresponding ThreadID - the supplied LogView tool will handle it as expected. This can be very useful for a multi-threaded server process, e.g. as implement with mORMot's Client-Server classes Client-Server process (page 318).

### 25.2.7. Log to the console

For debugging purposes, it could be very handy to output the logging content to a console window. It enables interactive debugging of a Client-Server process, for instance: you can interact with the Client, then look in real time at the server console window, and inspect which requests are processed, without the need to open the log file.

The EchoToConsole property enables you to select which events are to be echoed on the console (perhaps you expect only errors to appear, for instance).

```plaintext
with TSQLLog.Family do begin
  Level := LOG_VERBOSE;
  EchoToConsole := LOG_VERBOSE;  // log all events to the console
end;
```

Depending on the events, colors will be used to write the corresponding information. Errors will be displayed as light red, for instance.

Note that this echoing process slow down the logging process a lot, since it is currently implemented in a blocking mode, and writing to the console under Windows is much slower than writing to a file. This feature is therefore disabled by default, and not to be enabled on a production server, but only to make interactive debugging easier.

### 25.2.8. Remote logging

By default, TSynLog writes its activity to a local file, and/or to the console. The log file can be transmitted later on (once compressed) to support, for further review and debugging. But sometimes, it may be handy to see the logging in real-time, on a remote computer.

You can enable such remote monitoring for a given TSynLog class, by adding the mORMotHTTPClient.pas unit in your use clause, then calling the following constructor:

```plaintext
TSQLHttpClient.CreateForRemoteLogging('192.168.1.15',SQLite3Log,'8091','LogService');
```

This command will let any SQLiteLog event be sent to a remote server running at http://192.168.1.15:8091/LogService/RemoteLog - in fact this should be a mORMot server, but may be any REST server, able to answer to a PUT command sent to this URI.

A TSQLHttpClient instance will be created, and will be managed by the SQLite3Log instance. It will be released when the application will be closed, or when the SQLite3Log.Family.EchoRemoteStop method will be called.

In practice, our Log View tool - see below (page 639) - is able to run as a compatible remote server. Execute the tool, set the expected Server Root name ('LogService' by default), and the expected Server Port (8091 by default), then click on the "Server Launch" button. The Log View tool will now display in real time all incoming events, search into their content, and allow to save all received events into a regular .log or .synlz file, for further archiving and study. Note that since the Log View tool will run a http.sys based server - see High-performance http.sys server (page 326) - you may have to run once the tool with administrator rights, to register the Server Root / Server Port combination for binding.
Implementation of this remote logging has been tuned on both client and server side.
On client side, log events are gathered and sent in a dedicated background thread: if a lot of events are generated, they will be transferred in chunks of several rows, to minimize resource and bandwidth. On server side, incoming events are stored in memory, and indexed on the fly, with a periodic refresh rate of 500 ms: even a very active client logger will just let the Log View tool be responsive and efficient.

Thanks to the nature of the http.sys based server, several Server Root URI can be accessed in parallel with several Log View tool instance, on the same HTTP port: it will ease the IT policy of your network, since a single forwarded port will be able to handle several incoming connections.

See the "RemoteLoggingTest.dpr" sample from "11 - Exception logging", in conjunction with the LogView.dpr tool available in the same folder, for a running example of remote logging.

Note that our cross-platform clients - see Cross-Platform clients (page 481) - are able to log to a remote server, with the same exact format as used by our TSynLog class.

### 25.2.9. Log to third-party libraries

Our TSynLog class was designed to write its information to a file, and optionally to the console or a remote log server (as we just saw). In fact, TSynLog is extensively used by the mORMot framework to provide various levels of details on what happens behind the scene: it is great for debugging purposes.

It may be convenient to let TSynLog work with any third party logging applications such as CodeSite or SmartInspect, or any proprietary solution. As a result, mORMot logs can be mixed with existing application logs.

You can define the TSynLogFamily.EchoCustom property to specify a simple event to be triggered for each log operation: the application can then decide to log to a third party logger application.

Note that there is also the TSynLogFamily.NoFile property, which allows to disable completely the built-in file logging mechanism.

For instance, you may write:

```pascal
procedure TMyClass.Echo(Sender: TTextWriter; Level: TSynLogInfo; const Text: RawUTF8);
begin
  if Level in LOG_STACKTRACE then // filter only errors
    writeln(Text); // could be any third-party logger
end;
```

A process similar to TSynLogFile.ProcessOneLine() could then parse the incoming Text value, if needed.

### 25.2.10. Automated log archival

Log archives can be created with the following settings:

```pascal
with TSynLogDB.Family do
begin
  (...)
```
OnArchive := EventArchiveZip;
ArchivePath := '\\Remote\WK52382\Archive\Logs'; // or any path
end;

The ArchivePath property can be set to several functions, taking a timeout delay from the ArchiveAfterDays property value:
- nil is the default value, and won't do anything: the .log will remain on disk until they will be deleted by hand;
- EventArchiveDelete in order to delete deprecated .log files;
- EventArchiveSynLZ to compress the .log file into a proprietary SynLZ format: resulting file name will be located in ArchivePath\log\YYYYMM\*.log.synlz, and the command-line UnSynLz.exe tool (calling FileUnSynLz function of SynCommons.pas unit) can be used to uncompress it in to plain .log file;
- SynZip.EventArchiveZip will archive the .log files in ArchivePath\log\YYYYMM.zip files, grouping every 

SynLZ files are less compressed, but created much faster than .zip files. However, .zip files are more standard, and on a regular application, compression speed won't be an issue for the application.

25.2.11. Log files rotation
You can set TSynLogFamily.RotateFileCount and RotateFileSizeKB properties, to enable log file rotation:
- If both values are > 0, the log file will have a fixed name, without any time-stamp within;
- RotateFileSizeKB will define the maximum size of the main uncompressed log file
- RotateFileCount will define how many files are kept on disk - note that rotated files are compressed using SynLZ, so compression will be very fast.

Log file rotation is as easy as:

```pascal
with TSQLLog.Family do begin
  Level := LOG_VERBOSE;
  RotateFileCount := 5;               // will maintain a set of up to 5 files
  RotateFileSizeKB := 20*1024;        // rotate by 20 MB logs
end;
```

Such a logging definition will create those files on disk, e.g. for the TestSQL3.dpr regression tests:
- TestSQL3.1.log which will be the latest (current) log file, uncompressed;
- TestSQL3.1.synlz to TestSQL3.4.synlz will be the 4 latest log files, after compression. Our Log Viewer tool - see below (page 639) - is able to uncompress those .synlz files directly.

Note that as soon as you active file rotation, PerThreadLog = ptOneFilePerThread and HighResolutionTimestamp properties will be ignored, since both features expect a single file to exist per TSynLog class.

As an alternative, or in addition to this by-size rotation pattern, you could specify a fixed time of the day to perform the rotation.
For instance, the following will perform automatic rotation of the log files, whatever their size, at 23:00 each evening:

```pascal
with TSQLLog.Family do begin
  Level := LOG_VERBOSE;
  RotateFileCount := 5;               // will maintain a set of up to 5 files
  RotateFileDailyAtHour := 23;        // rotate at 11:00 PM
end;
```

If the default behavior - which is to compress all rotated files into .synlz format, and delete the older files - does not fit your needs, you can set a custom event to the TSynLogFamily.OnRotate property,
which will take care of the file rotation process.

25.2.12. Integration within tests

Logging is integrated within the unit testing classes, so that any failure will create an entry in the log with the source line, and stack trace:

C:\Dev\lib\SQLite3\exe\TestSQL3.exe 0.0.0.0 (2011-04-13)
Host=Laptop User=MyName CPU=2*0-15-1827 OS=2.3=5.1.2600 Wow64=0 Freq=3579545
TSynLogTest 1.13 2011-04-13 05:40:25

20110413 05402559 fail TTestLowLevelCommon(00031D70) Low level common: TDynArray "dynamic array failure" stack trace 0002FE0B SynComm\ons.TDynArray.Init (15148) 00036736 SynCommons.Test64K (18206) 0003682F SynCommons.TTestLowLevelCommon._TDynArray (18214) 000E9C94 TestSQL3 (163)

The difference between a test suit without logging (TSynTests) and a test suit with logging (TSynTestsLogged) is only this overridden method:

```delphi
procedure TSynTestsLogged.Failed(const msg: string; aTest: TSynTestCase);
begin
  inherited;
  with TestCase[fCurrentMethod] do
  begin
    fLogFile.Log(sllFail, '%: % "%%", [Ident, TestName[fCurrentMethodIndex], msg], aTest);
  end;
end;
```

In order to enable tests logging, you have just to enable it, e.g. with:

```delphi
TSynLogTestLog.Family.Level := LOG_VERBOSE;
```

You can optionally redirect the following global variable at program initialization, to share testing events with the main mORMot logs:

```delphi
with TSQLLog.Family do begin
  Level := LOG_VERBOSE;
  TSynLogTestLog := TSQLLog;// share the same log file with whole mORMot
end;
```

25.2.13. Log Viewer

Since the log files tend to be huge (for instance, if you set the logging for our unitary tests, the 17,000,000 test cases do create a huge log file of about 550 MB), a log viewer was definitively in need.

The log-viewer application is available as source code in the "Samples" folder, in the "11 - Exception logging" sub-folder.

25.2.13.1. Open log files

You can run it with a specified log file on the command line, or use the "Browse" button to browse for a file. That is, you can associate this tool with your .log files, for instance, and you'll open it just by double-clicking on such files.

Note that if the file is not in our TSynLog format, it will still be opened as plain text. You'll be able to browse its content and search within, but all the nice features of our logging won't be available, of course.

It is worth saying that the viewer was designed to be fast.

In fact, it takes no time to open any log file. For instance, a 390 MB log file is opened in less than one second on my laptop. Under Windows Seven, it takes more time to display the "Open file" dialog window than reading and indexing the 390 MB content.

It uses internally memory mapped files and optimized data structures to access to the data as fast as
possible - see TSynLogFile class.

25.2.13.2. Log browser

The screen is divided into three main spaces:
- On the left side, the panel of commands;
- On the right side, the log events list;
- On the middle, an optional list of method calls, and another list of threads (not shown by default).

The command panel allows to Browse your disk for a .log file. This button is a toggle of an optional Drive / Directory / File panel on the leftmost side of the tool. When a .log / .synlz / .txt file is selected, its content is immediately displayed. You can specify a directory name as a parameter of the tool (e.g. in a .lnk desktop link), which will let the viewer be opened in "Browse" mode, starting with the specified folder.

A button gives access to the global Stats about its content (customer-side hardware and software running configuration, general numbers about the log), and even ask for a source code line number and unit name from a hexadecimal address available in the log, by browsing for the corresponding .map file (could be handy if you did not deliver the .map content within your main executable - which you should have to).

Just below the "Browse" button, there is an edit field available, with a ? button. Enter any text within this edit field, and it will be searched within the log events list. Search is case-insensitive, and was designed to be fast. Clicking on the ? button (or pressing the F3 key) allows to repeat the last search.

In the very same left panel, you can see all existing events, with its own color and an associated check-box. Note that only events really encountered in the .log file appear in this list, so its content will change between log files. By selecting / un-selecting a check-box, the corresponding events will be instantaneously displayed / or not on the right side list of events. You can right click on the events check-box list to select a predefined set of events.

The right colored event list follows the events appended to the log, by time order. When you click on an event, its full line content is displayed at the bottom on the screen, in a memo.

Having all SQL / NoSQL and Client-Server events traced in the log is definitively a huge benefit for customer support and bug tracking.

25.2.13.3. Customer-side profiler

One distinctive feature of the TSynLog logging class is that it is able to map methods or functions entering/leaving (using the Enter method), and trace this into the logs. The corresponding timing is also written within the "Leave" event, and allows application profiling from the customer side. Most of the time, profiling an application is done during the testing, with a test environment and database. But this is not, and will never reproduce the exact nature of the customer use: for instance, hardware is not the same (network, memory, CPU), nor the software (Operating System version, [anti-]virus installed)...

By enabling customer-side method profiling, the log will contain all relevant information. Those events are named "Enter" / "Leave" in the command panel check-box list, and written as + and - in the right-sided event list.

The "Methods profiler" options allow to display the middle optional method calls list. Several sort order are available: by name (alphabetical sort), by occurrence (in running order, i.e. in the same order than in the event log), by time (the full time corresponding to this method, i.e. the time written within the "Leave" event), and by proper time (i.e. excluding all time spent in the nested methods).
The "Merge method calls" check-box allows to regroup all identical method calls, according to their name. In fact, most methods are not called once, but multiple time. And this is the accumulated time spent in the method which is the main argument for code profiling.

I'm quite sure that the first time you'll use this profiling feature on a huge existing application, you'll find out some bottlenecks you will have never thought about before.

25.2.13.4. Per-thread inspection

If the TSynLog family has specified PerThreadLog := ptIdentifiedInOnFile property, a new column will be added for each log row, with the corresponding ThreadID of the logged action.

The log-viewer application will identify this column, and show a "Thread" group below the left-side commands. It will allow to go to the next thread, or toggle the optional Thread view list. By checking / un-checking any thread of this list, you are able to inspect the execution log for a given process, very easily. A right-click on this thread list will display a pop-up menu, allowing to select all threads or no thread in one command.

25.2.13.5. Server for remote logging

As was stated above, Remote logging (page 636) can use our Log View tool as server and real-time viewer for any remote client, either using TSynLog, or any cross-platform client - see Cross-Platform clients (page 481).

Using a remote logging is specially useful from mobile applications (written with Delphi / FireMonkey or with Smart Mobile Studio / AJAX). Our viewer tool allows efficient live debugging of such platforms.

25.2.14. Framework log integration

The framework makes an extensive use of the logging features implemented in the SynLog.pas unit - see Enhanced logging (page 631).

In its current implementation, the framework is able to log on request:
- Any exceptions triggered during process, via s11Exception and s11ExceptionOS levels;
- Client and server RESTful URL methods via s11C1ient and s11Server levels;
- SQL executed statements in the SQLite3 engine via the s11SQL level;
- JSON results when retrieved from the SQLite3 engine via the s11Result level;
- Main errors triggered during process via s11Error level;
- Security User authentication and session management via s11UserAuth;
- Some additional low-level information via s11Debug, s11Warning and s11Info levels.

Those levels are available via the TSQLLog class, inheriting from TSynLog, as defined in mORMot.pas.

Three main TSynLogClass global variables are defined in order to use the same logging family for the whole framework. Since mORMot units are decoupled (e.g. Database or ORM/SOA), several variables have been defined, as such:
- SynDBLog for all SynDB* units, i.e. all generic database code;
- SQLite3Log for all mORMot* units, i.e. all ORM related code;
- SynSQLite3Log for the SynSQLite3 unit, which implements the SQLite3 engine itself.

By default, redirection to the main TSQLLog class is done if you use some features within mORMot:
- mORMot.pas unit will define SQLite3Log as its own TSQLLog class;
- mORMotDB.pas unit initialization will set SynDBLog := TSQLLog;
- moRMotSQLite3.pas unit initialization will set SynSQLite3Log := TSQLLog.

You can set your own class type to SynDBLog / SynSQLite3Log if you expect separated logging.

As a result, if you execute the following statement at the beginning of TestSQL3.dpr, regression tests will produce some logging, and resulting into more than 740 MB of log file content, if executed:

```delphi
tSynLogTestLog := TSQLLog; // share the same log file with whole mORMot
with TSQLLog.Family do begin
  Level := LOG_VERBOSE;
  HighResolutionTimestamp := true;
  PerThreadLog := ptIdentifiedInOnFile;
end;
```

Creating so much log content won't increase the processing time much. On a recent laptop, whole regression tests process will spent only 2 seconds to write the additional logging, which is the bottleneck of the hard disk writing.

If logging is turned off, there is no speed penalty noticeable.

Logging could be very handy for interactive debug of a client application. Since our TSynLog / TSQLLog class feature an optional output to a console, you are able to see in real-time the incoming requests - see for instance how 14 - Interface based services\Project14ServerHttp.pas sample is initialized:

```delphi
begin
  // define the log level
  with TSQLLog.Family do begin
    Level := LOG_VERBOSE;
    EchoToConsole := LOG_VERBOSE; // log all events to the console
  end;
  // create a Data Model
  aModel := TSQLModel.Create([], ROOT_NAME);
  (....)
```

Of course, this interactive console refresh slows down the process a lot. It is therefore to be defined only for debugging purposes, not on production.
26. Source code

26.1. License

26.1.1. Three Licenses Model

The framework source code is licensed under a disjunctive three-license giving the user the choice of one of the three following sets of free software/open source licensing terms:

- Mozilla Public License, version 1.1 or later (MPL);
- GNU General Public License, version 2.0 or later (GPL);
- GNU Lesser General Public License, version 2.1 or later (LGPL), with linking exception of the FPC modified LGPL.

FPC modified LGPL is the Library GNU General Public License with the following modification:

As a special exception of the LGPL, the copyright holders of this library give you permission to link this library with independent modules to produce an executable, regardless of the license terms of these independent modules, and to copy and distribute the resulting executable under terms of your choice, provided that you also meet, for each linked independent module, the terms and conditions of the license of that module. An independent module is a module which is not derived from or based on this library. If you modify this library, you may extend this exception to your version of the library, but you are not obligated to do so. If you do not wish to do so, delete this exception statement from your version.

This allows the use of the framework code in a wide variety of software projects, while still maintaining intellectual rights on library code.

In short:
- For GPL projects, use the GPL license - see [http://www.gnu.org/licenses/gpl-2.0.html](http://www.gnu.org/licenses/gpl-2.0.html).
- For LGPL projects, use the LGPL license - see [http://www.gnu.org/licenses/lgpl-2.1.html](http://www.gnu.org/licenses/lgpl-2.1.html).
- For commercial projects, use the MPL License - see [http://www.mozilla.org/MPL/MPL-1.1.html](http://www.mozilla.org/MPL/MPL-1.1.html) - which is the most permissive, or the FPC modified LGPL license, thanks to its linking exception - see [http://wiki.freepascal.org/modified_LGPL](http://wiki.freepascal.org/modified_LGPL).

### 26.1.2. Publish modifications and credit for the library

In all cases, any modification made to this source code **should** be published by any mean (e.g. a download link), even in case of MPL. If you need any additional feature, use the forums and we may introduce a patch to the main framework trunk.

You do not have to pay any fee for using our MPL/GPL/LGPL libraries.

But please do not forget to put somewhere in your credit window or documentation, a link to [https://synopse.info](https://synopse.info), if you use any of the units published under this tri-license.

For instance, if you select the MPL license, here are the requirements:

- You accept the license terms with no restriction - see [http://www.mozilla.org/MPL/2.0/FAQ.html](http://www.mozilla.org/MPL/2.0/FAQ.html) for additional information;
- You have to publish any modified unit (e.g. `SynTaskD1alog.pas`) in a public web site (e.g. [http://SoftwareCompany.com/MPL](http://SoftwareCompany.com/MPL)), with a description of applied modifications, and no removal of the original license header in source code;
- You make appear some notice available in the program (About box, documentation, online help), stating e.g. *This software uses some third-party code of the Synopse mORMot framework (C) 2020 Arnaud Bouchez - [https://synopse.info](https://synopse.info) - under Mozilla Public License 1.1; modified source code is available at [http://SoftwareCompany.com/MPL](http://SoftwareCompany.com/MPL).*

### 26.1.3. Derivate Open Source works

If you want to include part of the framework source code in your own open-source project, you may publish it with a comment similar to this one (as included in the great *DelphiWebScript* project by Eric Grange - [http://code.google.com/p/dwscript/](http://code.google.com/p/dwscript/)):

```{}
{ Will serve static content and DWS dynamic content via http.sys
  kernel mode high-performance HTTP server (available since XP SP2).
  See
  WARNING: you need to first register the server URI and port to the http.sys stack.
  That is, run the application at least once as administrator.

  Sample based on official mORMot's sample
  "SQLite3\Samples\09 - HttpApi web server\HttpApiServer.dpr"

  Synopse mORMot framework. Copyright (C) 2020 Arnaud Bouchez
  Synopse Informatique - [https://synopse.info](https://synopse.info)

  Original tri-license: MPL 1.1/GPL 2.0/LGPL 2.1

  You will need at least the following files from mORMot framework
  to be available in your project path:
  - SynCommons.pas
  - Synopse.inc
  - SynLZ.pas
}
- SynZip.pas
- SynCrtSock.pas
- SynWinSock.pas
  https://synopse.info/fossil/wiki?name=Downloads

Note that this documentation is under GPL license only, as stated in this document front page.

26.1.4. Commercial licenses

Even though our libraries are Open Source with permissive licenses, some users want to obtain a license anyway. For instance, you may want to hold a tangible legal document as evidence that you have the legal right to use and distribute your software containing our library code, or, more likely, your legal department tells you that you have to purchase a license.

If you feel like you really have to purchase a license for our libraries, Synopse, the company that employs the architect and principal developer of the library, will sell you one. Please contact us directly for a contract proposal.
26.2. Availability

As a true Open Source project, all source code of the framework is available, and latest version can be retrieved from our online repository at https://synopse.info/fossil..

As an alternative, you can monitor or fork our projects from our GitHub repository, at http://github.com/synopse/mORMot..

The source has been commented following the scheme used by our SynProject documentation tool. That is all interface definition of the units have special comments, which were extracted then incorporated into this Software Architecture Design (SAD) document, in the following pages.

26.2.1. Obtaining the Source Code

Each official release of the framework is available in a dedicated SynopseSQLite3.zip archive from the official https://synopse.info.. web site, but you may want to use the latest version available.

The easiest is to download a nightly-generated archive of the latest version of the trunk, from https://synopse.info/files/mORMotNightlyBuild.zip..

As an alternative, you can manually obtain a .zip archive containing a snapshot of the latest version of the whole source code tree directly from this repository.

Follow these steps:
- Pointer your web browser at https://synopse.info/fossil..
- Click on the "Login" menu button.
- Log in as anonymous. The password is shown on screen. Just click on the "Fill out captcha" button then on the "Login" button. The reason for requiring this login is to prevent spiders from walking the entire website, downloading ZIP archives of every historical version, and thereby soaking up all our bandwidth.
- Click on the Timeline or Leaves link at the top of the page. Preferred way is Leaves which will give you the latest available version.
- Select a version of the source code you want to download: a version is identified by an hexadecimal link (e.g. 6b684fb2). Note that you must successfully log in as "anonymous" in steps 1-3 above in order to see the link to the detailed version information.
- Finally, click on the "Zip Archive" link, available at the end of the "Overview" header, right ahead to the "Other Links" title. This link will build a .zip archive of the complete source code and download it to your browser.

26.2.2. Expected compilation platform

The framework source code tree will compile and is tested for the following platforms:
- Delphi 6 up to Delphi 10.3 Rio compiler and IDE, with FreePascal Compiler (FPC) 3.x and Lazarus support;
- Server side on Windows 32-bit and 64-bit platforms (FPC or Delphi XE2 and up expected when targeting Win64);
- Linux 32-bit and 64-bit platform for servers using the FPC 3.2 fixes branch - now stable and tested in production since years (especially Debian/Ubuntu on x86_64);
- VCL client on Win32/Win64 - GUI may be compiled optionally with third-party non Open-Source TMS Components, instead of default VCL components - see http://www.tmssoftware.com/site/tmspack.asp..
Delphi FMX / FreePascal FCL cross-platform support (page 483) clients on any supported platforms;
- Smart Mobile Studio support (page 486) startup with 2.1, for creating AJAX / JavaScript / HTML5 / Mobile clients.

Some part of the library (e.g. SynCommons.pas, SynTests.pas, SynLog.pas SynPDF.pas or the External SQL database access (page 239) units) are also compatible with Delphi 5.

If you want to compile mORMot unit into packages, to avoid an obfuscated [DCC Error] E2201 Need imported data reference (SG) to access 'VarCopyProc' error at compilation, you should defined the USEPACKAGES conditional in your project's options. Open SynCommons.inc for a description of this conditional, and all over definitions global to all mORMot units - see SynCommons unit (page 104). To avoid related E1025 Unsupported language feature: 'Object' compilation error, you should probably also set "Generate DCUs only" in project's options "C/C++ output file generator".

The framework source code implementation and design tried to be as cross-platform and cross-compiler as possible, since the beginning. It is a lot of work to maintain compatibility towards so many tools and platforms, but we think it is always worth it - especially if you try not depend on Delphi only, which as shown some backward compatibility issues during its lifetime.

For HTML5 and Mobile clients, our main platform is Smart Mobile Studio, which is a great combination of ease of use, a powerful SmartPascal dialect, small applications (much smaller than FMX), with potential packaging as native iOS or Android applications (via PhoneGap).

The latest versions of the FreePascal Compiler together with its great Lazarus IDE, are now very stable and easy to work with. We don't support CodeTyphon, since we found some licensing issue with some part of it (e.g. Orca GUI library origin is doubtful). So we recommend using fpcupdeluxe - see below (page 655) - which is maintained by Alfred, a mORMot contributor. This is amazing to build the whole set of compilers and IDE, with a lot of components, for several platforms (this is a cross-platform project), just from the sources. I like Lazarus stability and speed much more than Delphi (did you ever tried to browse and debug included $I ... files in the Delphi IDE? with Lazarus, it is painless), even if the compiler is slower than Delphi's, and if the debugger is less integrated and even more unstable than Delphi's under Windows (yes, it is possible!). At least, it works, and the Lazarus IDE is small and efficient. Official Linux support is available for mORMot servers, with full features in the FPC 3.2 branch - we use it on producing on Linux 64-bit since years.

26.2.3. SQLite3 static linking for Delphi and FPC

Preliminary note: if you retrieved the source code from https://github.com/synopse/mORMot, you will have all the needed .obj/.o static files available in the expected folders. Just ignore this chapter.

In order to maintain our https://synopse.info/fossil/timeline source code repository in a decent size, we excluded the sqlite3.obj/.o storage in it, but provide the full source code of the SQLite3 engine in a custom sqlite3.c file, ready to be compiled with all conditional defined as expected by SynSQLite3Static.pas. You need to add the official SQLite3 amalgamation file from https://www.sqlite.org/download.html, and put its content into a SQLite3\amalgamation sub-folder, for proper compilation. Our custom sqlite3.c file will add encryption feature to the engine. Also look into SynSQLite3Static.pas comments if there is any manual patch needed for proper compilation of the amalgamation source.

Of course, you are not required to do the compilation: sqlite3.obj (for Delphi Win32) and sqlite3.o files (for Delphi Win64) are available for Delphi, as a separated download, from https://synopse.info/files/sqlite3obj.7z.
For Delphi, please download the latest compiled version of these .obj/.o files from this link. You can also use the supplied c.bat and c64.bat files to compile from the original sqlite3.c file available in the repository, if you have the bcc32/bcc64 C command-line compiler(s) installed.

For Win32, the free version works and was used to create the .obj file, i.e. C++Builder Compiler (bcc compiler) free download - as available from Embarcadero web site.

For native Windows 64-bit applications (since Delphi XE2), a sqlite3.o static file is also available from the same archive. If you need an external dynamic .dll for Win64, since there is no official SQLite3 download for Win64 yet, you can use the one we supply at https://synopse.info/files/SQLite3-64.7z..

For FPC, you need to download static .o files from https://synopse.info/files/sqlite3fpc.7z.. then uncompress the embedded static folder and its sub-folders at the mORMot root folder (i.e. where Synopse.inc and SynCommons.pas stay). If you retrieved the source code from our GitHub repository at https://github.com/synopse/mORMot.. you already got the static sub-folder as expected by the framework. Those static files have been patched to support optional encryption of the SQLite3 database file. Then enable the FPCSQLITE3STATIC conditional in your project, or directly modify Synopse.inc to include it, so that those .o files will be statically linked to the executable.

You could also compile the static libraries from the sqlite3.c source, to run with FPC - do not forget to enable the FPCSQLITE3STATIC conditional in this case also.

Under Windows, ensure the MinGW compiler is installed, then execute c-fpcmingw.bat from the SQLite3 folder. It will create the sqlite3.o and sqlite3fts.o files, as expected by FPC.

Under Linux, Use the c-fpcgccclin.sh bash script.

26.2.4. SpiderMonkey library

To enable JavaScript support in mORMot, we rely on our version of the SpiderMonkey library. See Scripting Engine (page 561).

You can download the needed files from https://synopse.info/files/synsm.7z..

Do not forget to copy both files in the executable folder. For instance, put both mozjs.dll and nspr4.dll files with your JSHttpApiServer.exe.

By now, this library will work only under Win32, with Delphi as compiler - it has not yet been tested nor ported to FPC and other platforms.

26.2.5. Folder layout

As retrieved from our source code repository, you'll find the following folder layout:
In fact, you will get:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>Root folder, containing common files</td>
</tr>
<tr>
<td>CrossPlatform</td>
<td>Contains code for cross-platform clients</td>
</tr>
<tr>
<td>HtmlView/</td>
<td>A fork of the freeware THtmlView component, used as a demo of the SynPdf unit - not finished, and not truly Unicode ready</td>
</tr>
<tr>
<td>LVCL/</td>
<td><em>Light VCL</em> replacement files for standard VCL (for Delphi 6-7 only)</td>
</tr>
<tr>
<td>RTL7/</td>
<td>Enhanced RTL .dcu for Delphi 7 (not mandatory at all), and FastMM4 memory manager to be used before Delphi 2006</td>
</tr>
<tr>
<td>SQLite3/</td>
<td>Contains all ORM / SOA related files of the framework (i.e. <em>mORMot</em> itself) and its documentation</td>
</tr>
<tr>
<td>SynDBDataset/</td>
<td>DB .pas-based external database providers</td>
</tr>
<tr>
<td>SynProject/</td>
<td>Source code of the <em>SynProject</em> tool, used to edit and generate this documentation</td>
</tr>
</tbody>
</table>

### 26.2.5.1. Root folder

In the *Root folder*, some common files are defined:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPort.*</td>
<td>A fork of the freeware <em>ComPort</em> Library ver. 2.63</td>
</tr>
<tr>
<td>PasZip.pas</td>
<td>ZIP/LZ77 Deflate/Inflate Compression in pure pascal (SynZip.pas is faster)</td>
</tr>
<tr>
<td>SynBigTable.pas</td>
<td>class used to store huge amount of data with fast retrieval</td>
</tr>
<tr>
<td>SynBz.pas</td>
<td>fast BZ2 compression/decompression</td>
</tr>
<tr>
<td>SynBzPas.pas</td>
<td>pascal implementation of BZ2 decompression</td>
</tr>
<tr>
<td>SynCommons.pas</td>
<td>common functions used by most Synopse projects</td>
</tr>
<tr>
<td>SynCrtSock.pas</td>
<td>classes implementing HTTP and WebSockets client and server protocol</td>
</tr>
<tr>
<td>SynBidirSock.pas</td>
<td>classes implementing HTTP and WebSockets client and server protocol</td>
</tr>
<tr>
<td>SynCrypto.pas</td>
<td>fast cryptographic routines (hashing and cypher)</td>
</tr>
<tr>
<td>SynDprUses.inc</td>
<td>generic header included in the beginning of the uses clause of a .dpr source code</td>
</tr>
<tr>
<td>SynEcc.pas</td>
<td>certificate-based public-key cryptography using ECC-secp256r1</td>
</tr>
<tr>
<td>SynGdiPlus.pas</td>
<td>GDI+ library API access with anti-aliasing drawing</td>
</tr>
<tr>
<td>SynLog.pas</td>
<td>logging functions used by most Synopse projects</td>
</tr>
<tr>
<td>File</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SynLizard.pas</td>
<td>Lizard (LZ5) compression/decompression unit</td>
</tr>
<tr>
<td>SynLZ.pas</td>
<td>SynLZ compression/decompression unit - used by SynCommons.pas</td>
</tr>
<tr>
<td>SynLZO.pas</td>
<td>LZO compression/decompression unit</td>
</tr>
<tr>
<td>SynMemoEx.pas</td>
<td>Synopse extended TMemo visual component (used e.g. in SynProject) - for pre-Unicode Delphi only</td>
</tr>
<tr>
<td>SynMongoDB.pas</td>
<td>Direct MongoDB NoSQL database access</td>
</tr>
<tr>
<td>SynMustache.pas</td>
<td>Mustache logic-less template engine</td>
</tr>
<tr>
<td>SynPdf.pas</td>
<td>PDF file generation unit</td>
</tr>
<tr>
<td>SynScaleMM.pas</td>
<td>multi-thread friendly memory manager unit - not finished yet</td>
</tr>
<tr>
<td>SynSelfTests.pas</td>
<td>automated tests for mORMot Framework</td>
</tr>
<tr>
<td>SynSMAPI.pas</td>
<td>SpiderMonkey JavaScript engine API definition</td>
</tr>
<tr>
<td>SynSM.pas</td>
<td>SpiderMonkey JavaScript engine higher level classes</td>
</tr>
<tr>
<td>SynSQLite3.pas</td>
<td>SQLite3 embedded Database engine</td>
</tr>
<tr>
<td>SynSQLite3Static.pas</td>
<td>statically linked SQLite3 engine (with associated .obj/.o)</td>
</tr>
<tr>
<td>SynSSPIAuth.pas</td>
<td>low level access to Windows Authentication</td>
</tr>
<tr>
<td>SynTaskDialog.*</td>
<td>implement TaskDialog window (native on Vista/Seven, emulated on XP)</td>
</tr>
<tr>
<td>SynTests.pas</td>
<td>cross-compiler unitary tests functions</td>
</tr>
<tr>
<td>SynWinSock.pas</td>
<td>low level access to network Sockets for the Windows platform</td>
</tr>
<tr>
<td>SynZip.*</td>
<td>low-level access to ZLib compression, 1.2.5</td>
</tr>
<tr>
<td>SynZipFiles.pas</td>
<td>high-level access to .zip archive file compression</td>
</tr>
<tr>
<td>Synopse.inc</td>
<td>generic header to be included in all units to set some global conditional definitions</td>
</tr>
<tr>
<td>vista.*</td>
<td>A resource file enabling theming under XP</td>
</tr>
<tr>
<td>vistaAdm.*</td>
<td>A resource file enabling theming under XP and Administrator rights under Vista</td>
</tr>
</tbody>
</table>

In the same Root folder, the external database-agnostic units are located:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynDB.pas</td>
<td>abstract database direct access classes</td>
</tr>
<tr>
<td>SynOleDB.pas</td>
<td>fast OleDb direct access classes</td>
</tr>
<tr>
<td>SynDBODBC.pas</td>
<td>fast ODBC direct access classes</td>
</tr>
</tbody>
</table>
### 26.2.5.2. SynDBDataset folder

In a SynDBDataset folder, some external database providers are available, to be used with the SynDBDataset.pas classes:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynDBBDE.pas</td>
<td>BDE access classes</td>
</tr>
<tr>
<td>SynDBNexusDB.pas</td>
<td>NexusDB access classes</td>
</tr>
<tr>
<td>SynDBFireDAC.pas</td>
<td>FireDAC / AnyDAC library access classes</td>
</tr>
<tr>
<td>SynDBUniDAC.pas</td>
<td>UniDAC library access classes</td>
</tr>
</tbody>
</table>

### 26.2.5.3. SQLite3 folder

In the SQLite3/ folder, the files implementing the Synopses mORMot framework itself, i.e. its ORM and SOA features (using units from the Root folder):

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation/</td>
<td>Sub folder containing the source of the framework documentation</td>
</tr>
<tr>
<td>Samples/</td>
<td>Sub folders containing some sample code</td>
</tr>
<tr>
<td>mORMot.pas</td>
<td>Main ORM / SOA unit of the framework</td>
</tr>
<tr>
<td>mORMotDB.pas</td>
<td>Virtual Tables for ORM external SynDB.pas access</td>
</tr>
<tr>
<td>mORMotFastCgiServer.pas</td>
<td>FastCGI server - not fully tested</td>
</tr>
<tr>
<td>mORMotHttpClient.pas</td>
<td>HTTP/1.1 Client</td>
</tr>
<tr>
<td>mORMotHttpServer.pas</td>
<td>HTTP/1.1 Server</td>
</tr>
<tr>
<td>mORMotReport.pas</td>
<td>Integrated Reporting engine</td>
</tr>
<tr>
<td>mORMotService.pas</td>
<td>Stand-alone Service</td>
</tr>
<tr>
<td>mORMotSQLite3.pas</td>
<td>SQLite3 kernel bridge between mORMot.pas and SynSQLite3.pas</td>
</tr>
<tr>
<td>mORMoti18n.pas</td>
<td>internationalization (i18n) routines and classes</td>
</tr>
</tbody>
</table>
mORMotMongoDB.pas | ODM integration of MongoDB NoSQL database
---|---
mORMotMVC.pas | MVC classes to develop high performance Web Applications
mORMotToolBar.pas | ORM ToolBar User Interface generation
mORMotUI.* | Grid to display Database content
mORMotUIEdit.* | Record edition dialog, used to edit record content on the screen
mORMotUILogin.* | some common User Interface functions and dialogs
mORMotUIOptions.* | General Options setting dialog, generated from code
mORMotUIQuery.* | Form handling queries to a User Interface Grid, using our ORM RTTI to define search parameters and algorithms
mORMotVCL.pas | DB VCL dataset using TSQLTable/TSQLTableJSON data access
*.bmp *.rc | Resource files, compiled into *.res files
TestSQL3.dpr | Main testing program of the Synopse mORMot framework
mORMotSelfTests.pas | Run as administrator for TestSQL3 to use http.sys on Vista/Seven
TestSQL3Register.dpr | Source code of the SQLite3 embedded Database engine

testSQL3.bmp
testSQL3.rc

testSQL3.dpr
mORMotSelfTests.pas
TestSQL3Register.dpr
c.bat sqlite3.c

26.2.5.4. CrossPlatform folder

In a CrossPlatform folder, some source code is available, to be used when creating mORMot clients for compilers or platforms not supported by the main branch:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCrossPlatform.inc</td>
<td>Includes cross-platform and cross-compiler conditionals</td>
</tr>
<tr>
<td>SynCrossPlatformJSON.pas</td>
<td>Cross-platform JSON support for Delphi and FPC</td>
</tr>
<tr>
<td>SynCrossPlatformREST.pas</td>
<td>Main unit, handling secured ORM and SOA RESTful client</td>
</tr>
<tr>
<td>SynCrossPlatformCrypto.pas</td>
<td>SHA256 and crc32 algorithms, used for authentication</td>
</tr>
<tr>
<td>SynCrossPlatformSpecific.pas</td>
<td>System-specific functions, e.g. HTTP clients</td>
</tr>
</tbody>
</table>

See Cross-Platform clients (page 481) for more information.
26.3. Delphi Installation

Note: for FPC setup, see below (page 655).

To setup mORMot for Delphi 6 up to Delphi 10.3 Rio, you have two ways: either download the framework from archives, or clone our GitHub repository at https://github.com/synopse/mORMot..

26.3.1. Manual download

Download and uncompress the framework archives, including all sub-folders, into a local directory of your computer (for instance, D:\Dev\mORMot).

| Snapshot of the latest source code repository | https://synopse.info/files/mORMotNightlyBuild.zip. |
| for FPC: static .o files for Windows or Linux/BSD | https://synopse.info/files/sqlite3fpc.7z. |
| optional 64-bit SQLite3 external library | https://synopse.info/files.SQLite3-64.7z. |
| 32-bit SpiderMonkey library | https://synopse.info/files/synsm.7z. |

26.3.2. Get from GitHub

Or you may just clone our GitHub repository, from https://github.com/synopse/mORMot.. e.g. via:

```
d: cd Dev
git clone https://github.com/synopse/mORMot.git
```

It will create a d:\Dev\mORMot local folder, which will eventually be re-synchronized with the official sources. Advantage of cloning our GitHub repository is that it contains binaries for static linking, (SQLite3 and FPC specific), in a single step.

Just take care that if you downloaded some other library from Synopse (e.g. from https://github.com/synopse/SynPDF.. or https://github.com/synopse/dmustache).. you should better use the main https://github.com/synopse/mORMot.. only, which contains other projects, to avoid any version confusion. We have seen a lot of installation problems reported in our forum due to source code file collision from several repositories, not in the same revision.

26.3.3. Setup the Delphi IDE

To let your IDE know about mORMot source code, add the following paths to your Delphi IDE (in Tools/Environment/Library or Tools/Options/Language/Delphi Options/Library menu depending on
your Delphi version):
- Library path:
  {...existing path...};D:\Dev\mORMot;D:\Dev\mORMot\SQLite3;D:\Dev\mORMot\SynDBDataset
- Search path:
  {...existing path...};D:\Dev\mORMot;D:\Dev\mORMot\SQLite3;D:\Dev\mORMot\SynDBDataset

For any cross-platform client, do not forget to include the D:\Dev\mORMot\CrossPlatform to the Delphi or FreePascal IDE paths of the corresponding targets.

For Smart Mobile Studio, execute CopySynCrossPlatformUnits.bat to set the needed units in the IDE repository.

Note that before Delphi 2006, you will need to download and install FastMM4 heap memory manager - from http://sourceforge.net/projects/fastmm.. or from the D:\Dev\mORMot\RTL7 sub folder of our repository - for some samples to work (without it, mORMot units will work, but will be slower).

Starting with Delphi 2006, FastMM4 is already included within the system RTL, so you do not need to download it.

Open the TestSQL3.dpr program from the SQLite3 sub-folder. You should be able to compile it and run all regression tests on your computer.

If you want to run the tests with the fast http.sys kernel-based HTTP server, you'll need to compile and run (as administrator) TestSQL3Register.dpr once before launching TestSQL3.dpr.

Then open the *.dpr files, as available in the SQLite3\Samples sub-folder. You should be able to compile all sample programs, including SynFile.dpr in the MainDemo folder.

Enjoy!
26.4. FreePascal / Lazarus Installation

Note: see also Delphi Installation (page 653).

26.4.1. Possible targets

You can use the FreePascal Compiler (FPC) to (cross-)compile the mORMot framework source code, targeting the following CPU and OS combinations:

- i386-win32
- x86_64-win64
- i386-linux
- x86_64-linux
- i386-freebsd
- x86_64-freebsd
- i386-darwin
- x86_64-darwin
- arm-linux
- aarch64-linux

32-bit and 64-bit Windows and Linux platforms are the main supported targets, used in production since years. Others may need some enhancements, and you are free to contribute! mORMot has been reported to work on a Raspberry Pi running Linux, thanks to FPC abilities - and with good performance and stability.

Linux is a premium target for cheap and efficient server Hosting (page 536). Since mORMot has no dependency, installing a new mORMot server is as easy as copying its executable on a blank Linux host, then run it. No need to install any framework nor runtime. Even the SQLite3 engine will be statically linked on most platforms, as we provide up-to-date binaries in our repository. You could even use diverse operating systems (several Linux or Windows Server versions) in your mORMot servers farm, with minimal system requirements, and updates.

For proper FPC compilation, ensure you have the following settings to your project:

- Other unit files (-Fu):
  D:\Dev\mORMot;D:\Dev\mORMot\SQLite3;D:\Dev\mORMot\SQLite3\DDD\infra
- Include files (-Fi):
  $(ProjOutDir);D:\Dev\mORMot;D:\Dev\mORMot\SQLite3
- Libraries (-fI):
  D:\Dev\mORMot\static\$(TargetCPU)-$(TargetOS)

Replace D:\Dev\mORMot path by the absolute/relative folder where you did install the framework. In practice, a relative path (e.g. ..\..\mORMot) is preferred.

26.4.2. Setup your dedicated FPC / Lazarus environment with fpcupdeluxe

We currently use the FPC 3.2 fixes branch compiler, and the corresponding Lazarus IDE.

If you want to use TDocVariant custom variant type (page 112), ensure that your revision includes the fix for http://mantis.freepascal.org/view.php?id=26773 bug, i.e. newer than revision 28995 from 2014-11-05T22:17:54. This bug was not fixed in 2.6.4 branch, but any newer 3.x revision should be enough.

But since the FPC trunk may be unstable, we will propose to put in place a stable development
environment based on the FPC 3.2 branch to work with your mORMot-based projects. It may ease support and debugging.

For this task, don't download an existing binary release of FPC / Lazarus, but use the fpccupdeluxe tool, as published at http://wiki.freepascal.org/fpccupdeluxe - it will allow to build your environment directly from the sources, and install it in a dedicated folder. Several FPC / Lazarus installations, with dedicated revision numbers, may coexist on the same computer: just ensure you run Lazarus from the shortcut created by fpccupdeluxe.

- Download the latest release of the tool from https://github.com/LongDirtyAnimAlf/fpccupdeluxe/releases.
- Unpack it in a dedicated folder, and run its executable.
- On the main screen, locate on the left the two versions listboxes. Select "3.2" for FPC version and "2.0.10" for Lazarus version.
- Then build the FPC and Lazarus binaries directly from the latest sources, by clicking on "Install/update FPC+Laz".

Those branches are currently used for building our production projects, so are expected to be properly tested and supported.

At the time of the writing of this documentation, our Lazarus IDE (on Linux) reports using:
- FPC SVN 45643 (3.2.0)
- Lazarus SVN 63526 (2.0.10).

One big advantage of fpccupdeluxe is that you can very easily install cross-compilers for the CPU / OS combinations enumerated at Possible targets (page 655). Just go to the "Cross" tab, then select the target systems, and click on "Install compiler". It may be needed to download the cross-compiler binaries (once): just select "Yes" when prompted.

You could install mORMot using fpccupdeluxe, but we recommend you clone our https://github.com/synopse/mORMot repository, and setup the expected project paths, as detailed above at Delphi Installation (page 653).

If you don't want to define a given version, the current trunk should/could work, if it didn't include any regression at the time you get it - this is why we provide "supported" branches.

If you want to use the FPC trunk, please modify line #262 in Synopse.inc to enable the FPC_PROVIDE_ATTR_TABLE conditional and support the latest trunk RTTI changes:

```delfi
{$if not defined(VER3_0) and not defined(VER3_2) and not defined(VER2)}
  {Define FPC_PROVIDE_ATTR_TABLE} // to be defined since SVN 42356-42411
  // on compilation error in SynFPCTypInfo, undefine the above conditional
{$ifend}
```

Sadly, there is no official conditional available to have this RTTI change detected. You need to define globally this conditional.

### 26.4.3. Missing RTTI for interfaces in old FPC 2.6

Sadly, if you use a somewhat old revision of FPC, you may have to face some long-time unresolved FPC compiler-level restriction/issue, which did not supply the needed interface RTTI, which was available since Delphi 6 - see http://bugs.freepascal.org/view.php?id=26774.

As a consequence, SOA, mock/stub and MVC framework features will not work directly with older FPC revisions.

You could upgrade to a more recent FPC - we encourage you to Setup your dedicated FPC / Lazarus environment with fpccupdeluxe (page 655) - or we will propose here a workaround to compile such
mORMot applications with oldest FPC. The trick is to use Delphi to generate one unit containing the needed information.

The mORMotWrappers.pas unit proposes a ComputeFPCInterfacesUnit() function, which could be used on Delphi to generate the RTTI unit for FPC, as such:
- Ensure that the application will use all its needed interface: for instance, run all your regression tests, and/or use all its SOA/MVC features if you are not confident about your test coverage;
- Just before the application exits, add a call to ComputeFPCInterfacesUnit() with the proper folders, e.g. at the very end of your .dpr code.

For instance, here is how TestSQL3.dpr has been modified:

```delphi
program TestSQL3;
...
uses
  mORMotWrappers.pas,
...
begin
SQLite3ConsoleTests;
{$ifdef COMPUTEFPCINTERFACES}
ChDir(ExtractFilePath(ParamStr(0)));
ComputeFPCInterfacesUnit(
  ['..\CrossPlatform\templates', '..\..\CrossPlatform\templates'],
  '..\..\SQLite3\TestSQL3FPCInterfaces.pas'
);
{$endif}
end.
```

If you define the COMPUTEFPCINTERFACES conditional, the TestSQL3FPCInterfaces.pas unit will be generated.

Of course, for your own application, you may use absolute path names: here we used relative naming, via ..\, so that it will work on any development folder configuration.

Then, add it to any of your uses clause, as such:

```delphi
uses
  TestSQL3FPCInterfaces, // will register RTTI for interfaces under FPC
...
```

This unit will do nothing when compiled under Delphi: it will register the RTTI only when compiled with FPC.

The rest of your code will be untouched, and could be shared between Delphi and FPC.

If you do not modify the interface methods definition, this generation step could be safely bypassed.

We hope that in a close future, the FPC team will fix the [http://bugs.freepascal.org/view.php?id=26774](http://bugs.freepascal.org/view.php?id=26774) issue, but the ticket seems pretty inactive since its creation.

### 26.4.4. Writing your project for FPC

If you want your application to compile with FPC, some little patterns should be followed.

In all your source code file, the easiest is to including the following mORMot file, which will define all compiler options and conditionals as expected:

```delphi
{$I Synopse.inc} // define HASINLINE USETYPEINFO CPU32 CPU64 OWNORMTOUPPER
```

Then in your .dpr file, you should write:
uses
{$ifdef FPC} // we may be on Kylix or upcoming Delphi for Linux
{$ifdef Linux}
// if you use threads
ctthreads,
// widestring manager for Linux if needed !!
// could also be put in another unit ... but doc states: as early as possible
cstring, // optional
{$endif}
{$endif}

In fact, these above lines have been added to SynDprUses.inc, so you may just write the following:

uses
{$I SynDprUses.inc} // will enable FastMM4 prior to Delphi 2006, and enable FPC on Linux

As a side benefit, you will be able to share the same .dpr with Delphi, and it will enable FastMM4 for older versions which do not include it as default heap manager.

For instance a minimal FPC project to run the regression tests may be:

program LinuxSynTestFPCLinuxi386;

{$I Synopse.inc}
{$APPTYPE CONSOLE}

uses
{$I SynDprUses.inc}
mORMotSelfTests;

begin
SQLite3ConsoleTests;
end.

In your user code, ensure you do not directly link to the Windows unit, but rely on the cross-platform classes and functions as defined in SysUtils.pas, Classes.pas and SynCommons.pas. You could find in SynFPCTypInfo.pas and SynFPCLinux.pas some low-level functions dedicated to FPC and Linux compilation, to be used with legacy units - your new code should better rely on higher level functions and classes.

If you rely on mORMot classes and types, e.g. use RawUTF8 for all your string process in the business logic, and do not use Delphi-specific features (like generics, or new syntax sugar), it will be very easy to let your application compile with FPC.

### 26.4.5. Linux VM installation tips

Here are a few informal notes about getting running a FPC/Lazarus virtual machine running XUbuntu, on a Windows host. They are published as a general guideline, and we will not provide any reference procedure, nor support it. As stated in Setup your dedicated FPC / Lazarus environment with fpocupdeluxe (page 655), instead of using a virtual machine, you could just install the needed cross-compilers, then generate your Linux/BSD executables from your Windows Lazarus.

- Install the latest VirtualBox version from [http://www.virtualbox.org/](http://www.virtualbox.org/) to Windows;
- Download the latest .iso version published at [http://xubuntu.org/](http://xubuntu.org/), or any other place - we use XFCE since it is a very lightweight desktop, perfect to run Lazarus, and we selected an Ubuntu LTS revision (14.04 at the time of this writing), which will be the same used on Internet servers;
- Create a new virtual machine (VM) in VirtualBox, with 1 or 2 CPUs, more than 512 MB of RAM (we use 777 MB), and an automatic-growing disk storage, with a maximal size of 15 GB; ensure that the disk storage is marked as SSD if your real host storage is a SSD;
- Let the CDROM storage point to the .iso you downloaded;
- Start the VM and install Linux locally, as usual - you may select to download the updated packages during the installation, for safety;
- When the system restarts, if it asks for software updates, accept and wait for the update installation to finish - it is a good idea to have the latest version of the kernel and libraries before installing the VirtualBox drivers;
- Restart your VM when asked to;
- Under a Ubuntu/Debian terminal, write the following commands:

```bash
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install dkms
```
- Restart the VM, then select "Insert Guest Additions CD image" from the VM "Devices" menu: a virtual CD will be mounted on your system and appear on your desktop;
- Run the following command, according to your current user name and VirtualBox version:

```bash
sudo sh /media/...user.../VBOXADDITONS_....VBoxLinuxAdditions.run
```
- Restart the VM, then add a permanent shared folder in the VM configuration, named Lib, and pointing to your local mORMot installation (e.g. d:\Dev\mORMot;)
- Create a void folder, e.g. in your home:

```bash
mkdir lib
```
- Create a launcher for the following command, to mount the shared folder as expected:

```bash
sudo mount -t vboxsf lib /home/...user.../lib
```
- Execute the following commands:

```bash
sudo apt-get install build-essential mingw32-binutils subversion libgtk2.0-dev
sudo ln -s /usr/bin/i586-mingw32msvc-windres /usr/bin/windres
```
- Then install FPC / Lazarus as detailed in Setup your dedicated FPC / Lazarus environment with fp
cupdeluxe (page 655)
- If you have issues during SVN retrieval, go the development/fpc folder, then run the following before trying again the fpcup_linux_x86 command:

```bash
svn cleanup
svn update
```

If you followed the above steps, you should now have the expected Lazarus IDE and the corresponding FPC compiler. It is amazing seeing the whole compiler + IDE being compiled from the official sources, for free, and in a few minutes.

### 26.5. CrossKylix support

#### 26.5.1. What is Cross-Kylix?

The framework source code can also be cross-compiled under Delphi into a Linux executable, using CrossKylix.

https://crosskylix.untergrund.net.. is a free toolkit to integrate the Borland Kylix (Delphi for Linux) compiler into the Delphi Windows IDE.

CrossKylix has indeed several known drawbacks:

- It is a dead project, but an alive product. It still works!
- You can not buy it any more. Kylix 3 was shipped with Delphi 7.
- You need an actual Kylix CD (or an ISO image) to install it, since CrossKylix is just a wrapper around the official compiler, to let it run under Windows.
- Visual applications (based on the CLX framework - the predecessor of FMX) may still compile, but should not be used. But for server applications, it is still a pretty viable solution.
- The debugger and IDE is unusable. But thanks to our SynLog.pas you can debug your applications, with a full stack trace in the log, in case of any exception.

We added CrossKylix support for several reasons:
- We use it since years, with great success, so we know it better than FPC.
- It has still a better compiler than FPC, e.g. for the RTTI we need on interfaces, or even for executable size and memory use.
- Its compilation is instant - whereas FPC is long to compile.
- It supports FastMM4, which performs better than the FPC memory manager, from our tests.
- Resulting executables, for mORMot purpose, are faster than FPC - timing based on the regression tests.
- If the code works with Delphi 7, it will certainly work with Kylix (since it shares the same compiler and RTL), whereas FPC is compatible, but not the same. In particular, it does not suffer from limited RTTI or other FPC limitations. So it sounds safer to be used on production than FPC, even today.
- There is not a lot ofIFDEF, but in SynCommons.pas. Then there is a SynKylix.pas unit for several functions. User code will be the same than Delphi and FPC.
- There is a Linux compiler just released by Embarcadero since latest Delphi, but an Enterprise license is required, so we currently skip its support, and focus on FPC...

Currently, we use FPC with success for building i386 and x86_64 executables. FPC is therefore recommended for production work targeting Linux. See FreePascal / Lazarus Installation (page 655) and Setup your dedicated FPC / Lazarus environment with fpcupdeluxe (page 655).

Once you have installed CrossKylix, and set up its search path to the same as Delphi - see Delphi Installation (page 653), you should be able to compile your project for Linux, directly from your Delphi IDE. Then you need an actual Linux system to test it - please check the Linux VM installation tips (page 658).

A minimal console application which will compile for both Delphi and CrossKylix, running all our regression tests, may be:

```pascal
program Test;

{$APPTYPE CONSOLE}

uses
  FastMM4, // optional - only for CrossKylix or Delphi < 2006
  mORMotSelfTests;

begin
  SQLite3ConsoleTests;
end.
```

Similar guidelines as for Writing your project for FPC (page 657) do apply with CrossKylix. In particular, you should never use the Windows unit in your server code, but rely on the cross-platform classes and functions as defined in SysUtils.pas, Classes.pas and SynCommons.pas.

We did not succeed to have a static SQLite3 library linked by the Kylix compiler. It compiles about the .o format - sounds like if its linker expects a gcc2 format (which is nowadays deprecated), and does not accept the gcc3 or gcc4 generated binaries. So you need to install the sqlite3 as external library on your Linux.

On a 32-bit system, it is just a one line - depending on your distribution, here Ubuntu:

```
sudo apt-get install sqlite3
```

For a 64-bit system, you need to explicitly install the x86 32-bit version of SQLite3:
sudo apt-get install sqlite3:i386

or download and install manually packages for both modes:

sudo dpkg -i libsqlite3-0_3.8.2-1ubuntu2_amd64.deb libsqlite3-0_3.8.2-1ubuntu2_i386.deb

You could try to get the latest .deb from https://launchpad.net/ubuntu/vivid/i386/libsqlite3-0.. If you want to download and install manually a .deb for x86, please install both i386 and amd64 revisions with the same exact version at once, otherwise dpkg will complain.

If it may be of any help, here are the static dependencies listed on a running 64-bit Ubuntu system, on a CrossKylix compiled executable:

As you can see, there is a very few dependencies - then same as FPC's executable in fact, with the addition of the external 1ibsqlite3.so.0, which is statically linked to FPC's version.

### 26.5.2. Running Kylix 32-bit executables on 64-bit Linux

For Ubuntu versions above 13.10, if you installed a 64-bit distribution, 32-bit executables - as generated by CrossKylix - may not be recognized by the system. Of course, we recommend using FPC (cross-)compiler, and build your executable natively for the x86_64-linux target.

In order to install the 32-bit libraries needed by mORMot 32-bit executables compiled by Kylix on Linux, please execute:

```
sudo apt-get install lib32z1 lib32ncurses5 lib32bz2-1.0
```

If you want SynCrtSock.pas to be able to handle https:// on a 64-bit system - e.g. if you want to run the TestSQL3 regression tests which download some json reference file over https - you will need also to install libcurl (and OpenSSL) in 32-bit, as such:

```
sudo apt-get install libcurl3:i386
```

If it may be for any help, here are the static dependencies listed on a running 64-bit Ubuntu system, on a FPC 3.2 compiled executable:

There is almost no dependency: installing a mORMot server under Linux is just as simple as copying an executable on a minimal blank Linux server. You do not need any LAMP runtime, virtual machine, installing other services, or execution environment.

Of course, you may better add a reverse proxy like nginx in front of your mORMot servers when connected on the Internet, but for a cloud-based solution, or a self-hosted office server, software requirements are pretty low.

### 26.6. Upgrading from a 1.17 revision
If you are upgrading from an older revision of the framework, your own source code should be updated.

For instance, some units where renamed, and some breaking changes introduced by enhanced features. As a consequence, a direct update is not possible.

To properly upgrade to the latest revision:

1. Erase or rename your whole previous \mORMot directory.

2. Download latest 1.18 revision files as stated just above.

3. Change your references to mORMot units:
   - Add in your uses clause SynTests.pas if you use testing features;
   - Add in your uses clause SynLog.pas if you use logging features;
   - Rename in your uses clauses any SQLite3Commons reference into mORMot.pas;
   - Rename in your uses clauses any SQLite3 reference into mORMotSQLite3.pas;
   - Rename in your uses clauses any other SQLite3* reference into mORMot*;
   - Add in one of your uses clause a reference to the SynSQLite3Static.pas unit (for Win32 or Linux).

4. Consult the units' headers about 1.18 for breaking changes, mainly:
   - Introducing TID = type Int64 as TSQLRecord.ID primary key, TIDDynArray as an array, and TRecordReference now declared as Int64 instead of plain IntPtr / integer;
   - Renamed Iso8601 low-level structure as TTimeLogBits;
   - TJSONSerializerCustomWriter and TJSONSerializerCustomReader callbacks changed;
   - TSQLRestServerCallBackParams which is replaced by the TSQLRestServerURIContext class;
   - rmJSON* enumerates replaced by TSQLRestRoutingREST and TSQLRestRoutingJSON_RPC classes;
   - Changed 'ï' into '~' character for mORMoti18n.pas (formerly SQLite3i18n.pas) language files.

Most of those changes will be easily identified at compile time. But a quick code review, and proper regression tests at application level is worth considering.

Feel free to get support from our forum, if needed.
### 27. mORMot Framework source

#### 27.1. mORMot Framework used Units

The mORMot Framework makes use of the following units.

**Units located in the "Lib\" directory:**

<table>
<thead>
<tr>
<th>Source File Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PasZip</td>
<td>ZIP/LZ77 Deflate/Inflate Compression in pure pascal</td>
<td>674</td>
</tr>
<tr>
<td>SynBidirSock</td>
<td>Implements bidirectional client and server protocol, e.g. WebSockets</td>
<td>680</td>
</tr>
<tr>
<td>SynBigTable</td>
<td>Class used to store huge amount of data with fast retrieval</td>
<td>704</td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynCrtSock</td>
<td>Classes implementing TCP/UDP/HTTP client and server protocol</td>
<td>1083</td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
<tr>
<td>SynCurl</td>
<td>Curl library direct access classes</td>
<td>1200</td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td>SynDBDataset</td>
<td>DB.pas TDataset-based direct access classes (abstract TQuery-like)</td>
<td>1267</td>
</tr>
<tr>
<td>SynDBMidasVCL</td>
<td>Fill a VCL TClientDataset from SynDB data access</td>
<td>1271</td>
</tr>
<tr>
<td>SynDBODBC</td>
<td>ODBC 3.x library direct access classes to be used with our SynDB architecture</td>
<td>1274</td>
</tr>
<tr>
<td>SynDBOracle</td>
<td>Oracle DB direct access classes (via OCI)</td>
<td>1280</td>
</tr>
<tr>
<td>SynDBPostgres</td>
<td>PostgreSQL direct access classes for SynDB units (not DB.pas based)</td>
<td>1289</td>
</tr>
<tr>
<td>SynDBRemote</td>
<td>Remote access to any RDBMS via HTTP using our SynDB architecture</td>
<td>1293</td>
</tr>
<tr>
<td>Source File Name</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>SynDBSQLite3</td>
<td>SQLite3 direct access classes to be used with our SynDB architecture</td>
<td>1298</td>
</tr>
<tr>
<td>SynDBVCL</td>
<td>DB VCL read-only dataset from SynDB data access</td>
<td>1304</td>
</tr>
<tr>
<td>SynDBZeos</td>
<td>ZEOS 7.x direct access classes for SynDB units (not DB.pas based)</td>
<td>1307</td>
</tr>
<tr>
<td>SynEcc</td>
<td>Certificate-based public-key cryptography using ECC-secp256r1</td>
<td>1313</td>
</tr>
<tr>
<td>SynFast WideString</td>
<td>This unit will patch the System.pas RTL to use a custom NON OLE COMPATIBLE WideString type, NOT using the slow Windows API, but FastMM4 (without COW)</td>
<td>1349</td>
</tr>
<tr>
<td>SynGdiPlus</td>
<td>GDI+ library API access</td>
<td>1350</td>
</tr>
<tr>
<td>SynLizard</td>
<td>Lizard (LZ5) compression routines (statically linked for FPC)</td>
<td>1359</td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td>SynLZ</td>
<td>SynLZ Compression routines</td>
<td>1393</td>
</tr>
<tr>
<td>SynLZO</td>
<td>Fast LZO Compression routines</td>
<td>1395</td>
</tr>
<tr>
<td>SynMongoDB</td>
<td>MongoDB document-oriented database direct access classes</td>
<td>1396</td>
</tr>
<tr>
<td>SynMustache</td>
<td>Logic-less mustache template rendering</td>
<td>1450</td>
</tr>
<tr>
<td>SynOleDb</td>
<td>Fast OleDB direct access classes</td>
<td>1458</td>
</tr>
<tr>
<td>SynPdf</td>
<td>PDF file generation</td>
<td>1473</td>
</tr>
<tr>
<td>SynProtoRelay</td>
<td>Implements asynchronous safe WebSockets tunnelling</td>
<td>1521</td>
</tr>
<tr>
<td>SynProtoRTSPHTTP</td>
<td>Implements asynchronous RTSP stream tunnelling over HTTP</td>
<td>1526</td>
</tr>
<tr>
<td>SynSelfTests</td>
<td>Automated tests for common units of the Synopse mORMot Framework</td>
<td>1528</td>
</tr>
<tr>
<td>SynSM</td>
<td>Features JavaScript execution using the SpiderMonkey library</td>
<td>1554</td>
</tr>
<tr>
<td>SynSMAPI</td>
<td>SpiderMonkey *.h header port to Delphi</td>
<td>1573</td>
</tr>
<tr>
<td>SynSQLite3</td>
<td>SQLite3 Database engine direct access</td>
<td>1646</td>
</tr>
<tr>
<td>SynSQLite3RegEx</td>
<td>REGEXP function for SQLite3 Database using PCRE library</td>
<td>1710</td>
</tr>
<tr>
<td>SynSQLite3Static</td>
<td>SQLite3 3.33.0 Database engine - statically linked for Windows/Linux</td>
<td>1711</td>
</tr>
<tr>
<td>Source File Name</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>SynSSPI</td>
<td>Low level access to Windows SSPI/SChannel API for the Win32/Win64 platform</td>
<td>1714</td>
</tr>
<tr>
<td>SynSSPIAuth</td>
<td>Low level access to Windows Authentication for the Win32/Win64 platform</td>
<td>1718</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td>SynTaskDialog</td>
<td>Implement TaskDialog window (native on Vista/Seven, emulated on XP)</td>
<td>1824</td>
</tr>
<tr>
<td>SynTests</td>
<td>Unit test functions used by Synopse projects</td>
<td>1832</td>
</tr>
<tr>
<td>SynVirtualDataSet</td>
<td>DB VCL read-only virtual dataset</td>
<td>1840</td>
</tr>
<tr>
<td>SynWinSock</td>
<td>Low level access to network Sockets for the Win32 platform</td>
<td>1843</td>
</tr>
<tr>
<td>SynZip</td>
<td>Low-level access to ZLib compression (1.2.5 engine version)</td>
<td>1845</td>
</tr>
<tr>
<td>SynZipFiles</td>
<td>High-level access to .zip archive file compression</td>
<td>1856</td>
</tr>
</tbody>
</table>
### Unit dependencies in the "Lib" directory

#### Units located in the "Lib\CrossPlatform" directory:

<table>
<thead>
<tr>
<th>Source File Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCrossPlatformCrypto</td>
<td>Cryptographic cross-platform units</td>
<td>1859</td>
</tr>
<tr>
<td>SynCrossPlatformJSON</td>
<td>Minimum stand-alone cross-platform JSON process using variants</td>
<td>1861</td>
</tr>
<tr>
<td>SynCrossPlatformREST</td>
<td>Minimum stand-alone cross-platform REST process for mORMot client</td>
<td>1870</td>
</tr>
<tr>
<td>SynCrossPlatformSpecific</td>
<td>System-specific cross-platform units</td>
<td>1892</td>
</tr>
<tr>
<td>SynCrossPlatformTests</td>
<td>Regression tests for mORMot's cross-platform units</td>
<td>1897</td>
</tr>
</tbody>
</table>

<img src="chart.png" alt="Chart showing unit dependencies">

#### Unit dependencies in the "Lib\SQLite3" directory:

<table>
<thead>
<tr>
<th>Source File Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td>mORMotDB</td>
<td>Virtual Tables for external DB access for mORMot</td>
<td>2276</td>
</tr>
<tr>
<td>mORMotDDD</td>
<td>Domain-Driven-Design toolbox for mORMot</td>
<td>2285</td>
</tr>
<tr>
<td>mORMotFastCgiServer</td>
<td>FastCGI HTTP/1.1 Server implementation for mORMot</td>
<td>2302</td>
</tr>
<tr>
<td>mORMotHttpClient</td>
<td>HTTP/1.1 RESTful JSON Client classes for mORMot</td>
<td>2306</td>
</tr>
<tr>
<td>mORMotHttpServer</td>
<td>HTTP/1.1 RESTFUL JSON Server classes for mORMot</td>
<td>2313</td>
</tr>
<tr>
<td>mORMoti18n</td>
<td>Internationalization (i18n) routines and classes for mORMot</td>
<td>2322</td>
</tr>
<tr>
<td>mORMotMidasVCL</td>
<td>Fill a VCL TClientDataset from TSQLTable/TSQLTableJSON data</td>
<td>2332</td>
</tr>
<tr>
<td>mORMotMongoDB</td>
<td>Direct optimized MongoDB access for mORMot's ORM</td>
<td>2334</td>
</tr>
<tr>
<td>mORMotMVC</td>
<td>Implements MVC patterns over mORMot's ORM/SOA and SynMustache</td>
<td>2339</td>
</tr>
<tr>
<td>Source File Name</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>mORMotReport</td>
<td>Reporting unit</td>
<td>2352</td>
</tr>
<tr>
<td>mORMotSelfTests</td>
<td>Automated tests for common units of the Synopse mORMot Framework</td>
<td>2369</td>
</tr>
<tr>
<td>mORMotService</td>
<td>Daemon management classes for mORMot, including low-level Win NT Service</td>
<td>2370</td>
</tr>
<tr>
<td>mORMotSQLite3</td>
<td>SQLite3 embedded Database engine used as the mORMot SQL kernel</td>
<td>2382</td>
</tr>
<tr>
<td>mORMotToolBar</td>
<td>ORM-driven Office 2007 Toolbar for mORMot</td>
<td>2390</td>
</tr>
<tr>
<td>mORMotUI</td>
<td>Grid to display database content for mORMot</td>
<td>2404</td>
</tr>
<tr>
<td>mORMotUIEdit</td>
<td>Record edition dialog, used to edit record content with mORMot</td>
<td>2415</td>
</tr>
<tr>
<td>mORMotUILogin</td>
<td>Some common User Interface functions and dialogs for mORMot</td>
<td>2418</td>
</tr>
<tr>
<td>mORMotUIOptions</td>
<td>General Options setting dialog for mORMot</td>
<td>2422</td>
</tr>
<tr>
<td>mORMotUIQuery</td>
<td>Form handling queries to a User Interface Grid for mORMot</td>
<td>2424</td>
</tr>
<tr>
<td>mORMotVCL</td>
<td>DB VCL dataset using TSQLTable/TSQLTableJSON data access</td>
<td>2426</td>
</tr>
<tr>
<td>mORMotWrappers</td>
<td>Generate cross-platform clients code and documentation from a mORMot server</td>
<td>2430</td>
</tr>
<tr>
<td>TestSQL3FPCInterfaces</td>
<td>SOA interface methods definition to circumvent FPC missing RTTI</td>
<td>2436</td>
</tr>
</tbody>
</table>
Units located in the "Lib\SQLite3\DDD\dom" directory:

<table>
<thead>
<tr>
<th>Source File Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dddDomAuthInterfaces</td>
<td>Shared DDD Domains: Authentication objects and interfaces</td>
<td>2437</td>
</tr>
<tr>
<td>dddDomCountry</td>
<td>Shared DDD Domains: TCountry object definition</td>
<td>2439</td>
</tr>
<tr>
<td>dddDomUserCQRS</td>
<td>Shared DDD Domains: User CQRS Repository interfaces</td>
<td>2442</td>
</tr>
<tr>
<td>dddDomUserInterfaces</td>
<td>Shared DDD Domains: User interfaces definition</td>
<td>2445</td>
</tr>
<tr>
<td>dddDomUserTypes</td>
<td>Shared DDD Domains: User objects definition</td>
<td>2447</td>
</tr>
</tbody>
</table>

Units located in the "Lib\SQLite3\DDD\infra" directory:

<table>
<thead>
<tr>
<th>Source File Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dddInfraApps</td>
<td>Shared DDD Infrastructure: Application/Daemon implementation classes</td>
<td>2450</td>
</tr>
<tr>
<td>dddInfraAuthRest</td>
<td>Shared DDD Infrastructure: Authentication implementation</td>
<td>2463</td>
</tr>
<tr>
<td>Source File Name</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td><code>dddInfraEmail</code></td>
<td>Shared DDD Infrastructure: implement an email validation service</td>
<td>2467</td>
</tr>
<tr>
<td><code>dddInfraEmailer</code></td>
<td>Shared DDD Infrastructure: generic emailing service</td>
<td>2471</td>
</tr>
<tr>
<td><code>dddInfraRepoUser</code></td>
<td>Shared DDD Infrastructure: User CQRS Repository via ORM</td>
<td>2475</td>
</tr>
<tr>
<td><code>dddInfraSettings</code></td>
<td>Shared DDD Infrastructure: Application/Daemon settings classes</td>
<td>2477</td>
</tr>
</tbody>
</table>
### Unit dependencies in the "Lib\SQLite3\DDD\infra" directory

Units located in the "Lib\SynDBDataset" directory:

<table>
<thead>
<tr>
<th>Source File Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynDBBDE</td>
<td>BDE access classes for SynDB units</td>
<td>2490</td>
</tr>
<tr>
<td>SynDBFireDAC</td>
<td>FireDAC/AnyDAC-based classes for SynDB units</td>
<td>2493</td>
</tr>
<tr>
<td>SynDBNexusDB</td>
<td>NexusDB 3.x direct access classes (embedded engine only)</td>
<td>2497</td>
</tr>
<tr>
<td>SynDBUniDAC</td>
<td>UniDAC-based classes for SynDB units</td>
<td>2501</td>
</tr>
</tbody>
</table>

![Diagram of unit dependencies]

---

*Unit dependencies in the "Lib\SynDBDataset" directory*
27.2. PasZip.pas unit

*Purpose:* ZIP/LZ77 Deflate/Inflate Compression in pure pascal
- this unit is a part of the freeware Synopse framework, licensed in the LGPL v3; version 1.18

**Objects implemented in the PasZip unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFileHeader</td>
<td>Directory file information structure, as used in .zip file format</td>
<td>675</td>
</tr>
<tr>
<td>TFileInfo</td>
<td>Generic file information structure, as used in .zip file format</td>
<td>674</td>
</tr>
<tr>
<td>TLastHeader</td>
<td>@TLocalFileHeader last header structure, as used in .zip file format</td>
<td>675</td>
</tr>
<tr>
<td>TLocalFileHeader</td>
<td>Internal file information structure, as used in .zip file format</td>
<td>675</td>
</tr>
<tr>
<td>TZipEntry</td>
<td>Stores an entry of a file inside a .zip archive</td>
<td>676</td>
</tr>
<tr>
<td>TZipRead</td>
<td>Read-only access to a .zip archive file</td>
<td>676</td>
</tr>
<tr>
<td>TZipWrite</td>
<td>Write-only access for creating a .zip archive file</td>
<td>677</td>
</tr>
</tbody>
</table>

**TFileInfo** = packed record

*Generic file information structure, as used in .zip file format*
- used in any header, contains info about following block

```pascal
extraLen: word;
  Length(name)
flags: word;
  14
zcrc32: dword;
  Dos format
zlastModDate: word;
  Dos format
zlastModTime: word;
  8 (deflate)
zzipMethod: word;
  0
```
TLocalFileHeader = packed record

*Internal file information structure, as used in*.zip* file format*
- used locally inside the file stream, followed by the name and then the data

    fileInfo: TFileInfo;
    04034b50

TFFileHeader = packed record

*Directory file information structure, as used in*.zip* file format*
- used at the end of the zip file to recap all entries

    extFileAttr: dword;
    0 = binary; 1 = text

    fileInfo: TFileInfo;
    14

    firstDiskNo: word;
    0

    intFileAttr: word;
    0

    localHeadOff: dword;

    madeBy: word;
    02014b50

TLastHeader = packed record

*@TLocalFileHeader last header structure, as used in*.zip* file format*
- this header ends the file and is used to find the TFFileHeader entries

    commentLen: word;
    @TFFileHeader

    headerDisk: word;
    0

    headerOffset: dword;

    headerSize: dword;
    1

    thisDisk: word;
    06054b50
thisFiles: word;
  0

totalFiles: word;
  1

**TZipEntry = packed record**

O stores an entry of a file inside a .zip archive

data: PAnsiChar;
  Points to the compressed data in the .zip archive, mapped in memory

info: PFileInfo;
  The information of this file, as stored in the .zip archive

Name: array[0..127 - SizeOf(pointer)*2] of AnsiChar;
  ASCIIZ name of the file inside the .zip archive
- not a string, but a fixed-length array of char

**TZipRead = class(TObject)**

Read-only access to a .zip archive file
- can open directly a specified .zip file (will be memory mapped for fast access)
- can open a .zip archive file content from a resource (embedded in the executable)
- can open a .zip archive file content from memory

Count: integer;
  The number of files inside a .zip archive

Entry: array of TZipEntry;
  The files inside the .zip archive

**constructor** Create(BufZip: pByteArray; Size: cardinal); overload;
  Open a .zip archive file directly from memory

**constructor** Create(Instance: THandle; const ResName: string; ResType: PChar); overload;
  Open a .zip archive file directly from a resource

**constructor** Create(const aFileName: TFileName; ZipStartOffset: cardinal = 0; Size: cardinal = 0; ShowMessageBoxOnError: boolean = true); overload;
  Open a .zip archive file as Read Only

**destructor** Destroy; override;
  Release associated memory

**function** CheckFile(aIndex: integer; DestPath: TFileName): boolean;
  Read the file from the supplied folder, and check its content according to the crc32 stored inside the .zip archive header (no decompression is made)

**function** GetInitialExeContent: RawByteZip;
  Get any initial .exe file
function NameToIndex(const aZipName: TZipName): integer;
    Get the index of a file inside the .zip archive

function UnZip(aIndex: integer): RawByteZip; overload;
    Uncompress a file stored inside the .zip archive into memory

function UnZipFile(aIndex: integer; DestPath: TFileName; ForceWriteFlush: boolean): boolean;
    Uncompress a file stored inside the .zip archive into a destination folder

property ZipStartOffset: cardinal read fZipStartOffset;
    The starting offset of the .zip content, after the initial .exe, if any
    - can be used to copy the initial .exe file

TZipWrite = class(TObject)
    Write-only access for creating a .zip archive file
    - not to be used to update a .zip file, but to create a new one
    - update can be done manually by using a TZipRead instance and the AddFromZip() method

    Count: integer;
        The total number of entries

    Entry: array of record name: TZipName; fhr: TFileHeader; end;
        The resulting file entries

    Handle: integer;
        The associated file handle

    constructor Create(const aFileName: TFileName); overload;
        The file name the corresponding file header initialize the .zip file

    destructor Destroy; override;
        Release associated memory, and close destination file

    procedure AddDeflated(const aFileName: TFileName; RemovePath: boolean = true; CompressLevel: integer = 6); overload;
        Compress (using the deflate method) a file, and add it to the zip file

    procedure AddDeflated(const aZipName: TZipName; Buf: pointer; Size: integer; CompressLevel: integer = 6; FileAge: integer = $1 + 1 shl 5 + 30 shl 9); overload;
        Compress (using the deflate method) a memory buffer, and add it to the zip file
        - by default, the 1st of January, 2010 is used if not date is supplied

    procedure AddFromZip(const ZipEntry: TZippedEntry);
        Add a file from an already compressed zip entry

    procedure AddStored(const aZipName: TZipName; Buf: pointer; Size: integer; FileAge: integer = $1 + 1 shl 5 + 30 shl 9);
        Add a memory buffer to the zip file, without compression
        - content is stored, not deflated (in that case, no deflate code is added to the executable)
        - by default, the 1st of January, 2010 is used if not date is supplied
procedure Append(const Content: RawByteZip);

Append a file content into the destination file
- useful to add the initial Setup.exe file, e.g.

Types implemented in the `PasZip` unit

PFileInfo = ^TFileInfo;

Functions or procedures implemented in the `PasZip` unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompressMem</td>
<td>Compress memory using the ZLib DEFLATE algorithm</td>
<td>678</td>
</tr>
<tr>
<td>CompressString</td>
<td>Compress memory using the ZLib DEFLATE algorithm with a crc32 checksum</td>
<td>678</td>
</tr>
<tr>
<td>CreateVoidZip</td>
<td>Create a void .zip file</td>
<td>678</td>
</tr>
<tr>
<td>GzCompress</td>
<td>Create a compatible .gz file (returns file size)</td>
<td>678</td>
</tr>
<tr>
<td>UnCompressMem</td>
<td>Uncompress memory using the ZLib INFLATE algorithm</td>
<td>678</td>
</tr>
<tr>
<td>UncompressString</td>
<td>Uncompress memory using the ZLib INFLATE algorithm, checking crc32 checksum</td>
<td>678</td>
</tr>
<tr>
<td>UpdateCrc32</td>
<td>Calculate the CRC32 hash of a specified memory buffer</td>
<td>678</td>
</tr>
<tr>
<td>Zip</td>
<td>You can create a &quot;zip&quot; compatible archive by calling the &quot;Zip&quot; function.</td>
<td>679</td>
</tr>
</tbody>
</table>

function CompressMem(src, dst: pointer; srcLen, dstLen: integer): integer;

Compress memory using the ZLib DEFLATE algorithm

function CompressString(const data: RawByteZip; failIfGrow: boolean = false): RawByteZip;

Compress memory using the ZLib DEFLATE algorithm with a crc32 checksum

procedure CreateVoidZip(const aFileName: TFileName);

Create a void .zip file

function GzCompress(src: pointer; srcLen: integer; const fName: TFileName): cardinal;

Create a compatible .gz file (returns file size)

function UnCompressMem(src, dst: pointer; srcLen, dstLen: integer): integer;

Uncompress memory using the ZLib INFLATE algorithm

function UncompressString(const data: RawByteZip): RawByteZip;

Uncompress memory using the ZLib INFLATE algorithm, checking crc32 checksum

function UpdateCrc32(aCRC32: cardinal; inBuf: pointer; inLen: integer): cardinal;

Calculate the CRC32 hash of a specified memory buffer
function Zip(const zip: TFileName; const files, zipAs: array of TFileName; NoSubDirectories: boolean = false): boolean;

You can create a "zip" compatible archive by calling the "Zip" function.
- The first parameter is the full file path of the new zip archive.
- The second parameter must be an array of the files you want to have zipped into the archive (full file path again, please).
- The third array (only file names, please) allows you to store the files into the zip under a different name.
- Generally the resulting zip archive should not contain any directory structure: all zipped files are directly stored in the archive's root, if NoSubDirectories is set to TRUE.

Variables implemented in the PasZip unit

crc32Tab: TCRC32Tab;

The static buffer used for fast CRC32 hashing
27.3. SynBidirSock.pas unit

*Purpose:* Implements bidirectional client and server protocol, e.g. WebSockets
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynBidirSock unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| SynCommons     | Common functions used by most Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 717  |
| SynCrtSock     | Classes implementing TCP/UDP/HTTP client and server protocol  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1083 |
| SynCrypto      | Fast cryptographic routines (hashing and cypher)  
- implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT  
- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1139 |
| SynEcc         | Certificate-based public-key cryptography using ECC-secp256r1  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1313 |
| SynLog         | Logging functions used by Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1363 |
| SynLZ          | SynLZ Compression routines  
- licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1393 |
| SynTable       | Filter/database/cache/buffer/security/search/multithread/OS features  
- as a complement to SynCommons, which tended to increase too much  
- licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1721 |
| SynWinSock     | Low level access to network Sockets for the Win32 platform  
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1843 |
### SynBidirSock class hierarchy

#### Objects implemented in the *SynBidirSock* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAsynchConnections</td>
<td>Exception associated with TAsynchConnection / TAsynchConnections process</td>
<td>684</td>
</tr>
<tr>
<td>EWebSockets</td>
<td>Exception raised when processing WebSockets</td>
<td>687</td>
</tr>
<tr>
<td>TAsynchClient</td>
<td>Implements thread-pooled high-performance TCP multiple clients</td>
<td>687</td>
</tr>
<tr>
<td>TAsynchConnection</td>
<td>Abstract class to store one TAsynchConnections connection</td>
<td>684</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TAsynchConnections</td>
<td>Implements an abstract thread-pooled high-performance TCP clients or server</td>
<td>685</td>
</tr>
<tr>
<td>TAsynchConnectionsSockets</td>
<td>Handle multiple non-blocking connections using TAsynchConnection instances</td>
<td>684</td>
</tr>
<tr>
<td>TAsynchConnectionsThread</td>
<td>Used to implement a thread poll to process TAsynchConnection instances</td>
<td>684</td>
</tr>
<tr>
<td>TAsynchServer</td>
<td>Implements a thread-pooled high-performance TCP server</td>
<td>686</td>
</tr>
<tr>
<td>THttpClientWebSockets</td>
<td>Socket API based REST and HTTP/1.1 client, able to upgrade to WebSockets</td>
<td>699</td>
</tr>
<tr>
<td>THttpRequestCache</td>
<td>In-memory storage of one THttpRequestCached entry</td>
<td>683</td>
</tr>
<tr>
<td>THttpRequestCached</td>
<td>Handles cached HTTP connection to a remote server</td>
<td>683</td>
</tr>
<tr>
<td>TWebCrtSocketProcess</td>
<td>TCrtSocket-based WebSockets process, used on both client or server sides</td>
<td>695</td>
</tr>
<tr>
<td>TWebSocketFrame</td>
<td>Stores a WebSockets frame</td>
<td>687</td>
</tr>
<tr>
<td>TWebSocketFrameList</td>
<td>Used to manage a thread-safe list of WebSockets frames</td>
<td>692</td>
</tr>
<tr>
<td>TWebSocketProcess</td>
<td>Abstract WebSockets process, used on both client or server sides</td>
<td>694</td>
</tr>
<tr>
<td>TWebSocketProcessClient</td>
<td>Implements WebSockets process as used on client side</td>
<td>699</td>
</tr>
<tr>
<td>TWebSocketProcessClientThread</td>
<td>WebSockets processing thread used on client side</td>
<td>699</td>
</tr>
<tr>
<td>TWebSocketProcessServer</td>
<td>Implements WebSockets process as used on server side</td>
<td>696</td>
</tr>
<tr>
<td>TWebSocketProcessSettings</td>
<td>Parameters to be used for WebSockets process</td>
<td>692</td>
</tr>
<tr>
<td>TWebSocketProtocol</td>
<td>One instance implementing application-level WebSockets protocol</td>
<td>688</td>
</tr>
<tr>
<td>TWebSocketProtocolBinary</td>
<td>Handle a REST application-level WebSockets protocol using compressed and optionally AES-CFB encrypted binary</td>
<td>690</td>
</tr>
<tr>
<td>TWebSocketProtocolChat</td>
<td>Simple chatting protocol, allowing to receive and send WebSocket frames</td>
<td>689</td>
</tr>
<tr>
<td>TWebSocketProtocolJSON</td>
<td>Handle a REST application-level WebSockets protocol using JSON for transmission</td>
<td>690</td>
</tr>
<tr>
<td>TWebSocketProtocolList</td>
<td>Used to maintain a list of websocket protocols (for the server side)</td>
<td>691</td>
</tr>
<tr>
<td>TWebSocketProtocolRest</td>
<td>Handle a REST application-level bi-directional WebSockets protocol</td>
<td>690</td>
</tr>
<tr>
<td>TWebSocketServer</td>
<td>Main HTTP/WebSockets server Thread using the standard Sockets API (e.g. WinSock)</td>
<td>697</td>
</tr>
</tbody>
</table>
TWebSocketServerResp
An enhanced input/output structure used for HTTP and WebSockets requests

TWebSocketServerRest
Main HTTP/WebSockets server Thread using the standard Sockets API (e.g. WinSock)

THttpRequestCache = record
In-memory storage of one THttpRequestCached entry

THttpRequestCached = class(TSynPersistent)
Handles cached HTTP connection to a remote server
- use in-memory cached content when HTTP_NOTMODIFIED (304) is returned for an already known ETAG header value

constructor Create(const aURI: RawUTF8; aKeepAliveSeconds: integer=30; aTimeoutSeconds: integer=15*60; const aToken: RawUTF8=''; aHttpClass: THttpRequestClass=nil); reintroduce;

Initialize the cache for a given server
- once set, you can change the request URI using the Address property
- aKeepAliveSeconds = 0 will force "Connection: Close" HTTP/1.0 requests
- an internal cache will be maintained, and entries will be flushed after aTimeoutSeconds - i.e. 15 minutes per default - setting 0 will disable the client-side cache content
- aToken is an optional token which will be transmitted as HTTP header:
  Authorization: Bearer <aToken>
- TWinHttp will be used by default under Windows, unless you specify another class

destructor Destroy; override;
Finalize the cache

function Flush(const aAddress: SockString): boolean;
Erase one resource from internal cache

function Get(const aAddress: SockString; aModified: PBoolean=nil; aStatus: PInteger=nil): SockString;
Retrieve a resource from the server, or internal cache
- aModified^ = true if server returned a HTTP_SUCCESS (200) with some new content, or aModified^ = false if HTTP_NOTMODIFIED (304) was returned

function LoadFromURI(const aURI: RawUTF8; const aToken: RawUTF8=''; aHttpClass: THttpRequestClass=nil): boolean;
Connect to a new server
- aToken is an optional token which will be transmitted as HTTP header:
  Authorization: Bearer <aToken>
- TWinHttp will be used by default under Windows, unless you specify another class

procedure Clear;
Finalize the current connection and flush its in-memory cache
- you may use LoadFromURI() to connect to a new server
property URI: TURI read fURI;
  Read-only access to the connected server

EAsynchConnections = class(ESynException)
  Exception associated with TAsynchConnection / TAsynchConnections process

TAsynchConnection = class(TSynPersistent)
  Abstract class to store one TAsynchConnections connection
  - may implement e.g. WebSockets frames, or IoT binary protocol
  - each connection will be identified by a TAsynchConnectionHandle integer
  - idea is to minimize the resources used per connection, and allow full customization of the
    process by overriding the OnRead virtual method (and, if needed, AfterCreate/AfterWrite/BeforeDestroy/OnLastOperationIdle)

  constructor Create(const aRemoteIP: RawUTF8); reintroduce; virtual;
    Initialize this instance

  property Handle: TAsynchConnectionHandle read fHandle;
    Read-only access to the handle number associated with this connection

  property RemoteIP: RawUTF8 read fRemoteIP;
    The associated remote IP4/IP6, as text

  property Socket: TSocket read fSlot.socket;
    Read-only access to the socket number associated with this connection

TAsynchConnectionsSockets = class(TPollAsynchSockets)
  Handle multiple non-blocking connections using TAsynchConnection instances
  - OnRead will redirect to TAsynchConnection.OnRead virtual method
  - OnClose will remove the instance from TAsynchConnections.fConnections[]
  - OnError will return false to shutdown the connection (unless acoOnErrorContinue is defined in
    TAsynchConnections.Options)

  function Write(connection: TObject; const data: dataslot; datalen: integer; timeout: integer=5000): boolean; override;
    Add some data to the asynchronous output buffer of a given connection
    - this overridden method will refresh TAsynchConnection.LastOperation
    - can be executed from an TAsynchConnection.OnRead method

  property Total: integer read GetTotal;
    How many clients have been handled by the poll, from the beginning

TAsynchConnectionsThread = class(TSynThread)
  Used to implement a thread poll to process TAsynchConnection instances

  constructor Create(aOwner: TAsynchConnections; aProcess: TPollSocketEvent);
    Initialize the thread
TAsynchConnections = class(TServerGeneric)

  Implements an abstract thread-pooled high-performance TCP clients or server
  - internal TAsynchConnectionsSockets will handle high-performance process of a high number of
    long-living simultaneous connections
  - will use a TAsynchConnection inherited class to maintain connection state
  - don't use this abstract class but either TAsynchServer or TAsynchClients
  - under Linux/POSIX, check your "ulimit -Hn" value: one socket consumes two file descriptors:
    you may better add the following line to your /etc/limits.conf or /etc/security/limits.conf system
    file:
    * hard nofile 65535

constructor Create(OnStart, OnStop: TNotifyThreadEvent; aStreamClass: TAsynchConnectionClass; const ProcessName: SockString; aLog: TSynLogClass; aOptions: TAsynchConnectionsOptions; aThreadPoolCount: integer); reintroduce;
  virtual;

    From fThreadClients initialize the multiple connections
    - warning: currently reliable only with aThreadPoolCount=1

destructor Destroy; override;

    Shut down the instance, releasing all associated threads and sockets

function ConnectionFindLocked(aHandle: TAsynchConnectionHandle; aIndex: PInteger=nil): TAsynchConnection;

    High-level access to a connection instance, from its handle
    - could be executed e.g. from a TAsynchConnection.OnRead method
    - returns nil if the handle was not found
    - returns the matching instance, and caller should release the lock as:
      try ... finally UnLock; end;

function ConnectionRemove(aHandle: TAsynchConnectionHandle): boolean;

    Remove an handle from the internal list, and close its connection
    - could be executed e.g. from a TAsynchConnection.OnRead method

function Write(connection: TAsynchConnection; const data: datalen: integer): boolean; overload;

    Add some data to the asynchronous output buffer of a given connection
    - could be executed e.g. from a TAsynchConnection.OnRead method

function Write(connection: TAsynchConnection; const data: SockString): boolean; overload;

    Add some data to the asynchronous output buffer of a given connection
    - could be executed e.g. from a TAsynchConnection.OnRead method

procedure Lock;

    Just a wrapper around fConnectionLock.Lock
procedure LogVerbose(connection: TAsynchConnection; const ident: RawUTF8; frame: pointer; framelen: integer); overload;

  Log some binary data with proper escape
  - can be executed from an TAsynchConnection.OnRead method to track content:
    if acoVerboseLog in Sender.Options then Sender.LogVerbose(self,...);

procedure LogVerbose(connection: TAsynchConnection; const ident: RawUTF8; const frame: RawByteString); overload;

  Log some binary data with proper escape
  - can be executed from an TAsynchConnection.OnRead method to track content:
    if acoVerboseLog in Sender.Options then Sender.LogVerbose(...);

procedure Unlock;

  Just a wrapper around fConnectionLock.UnLock

property Clients: TAsynchConnectionsSockets read fClients;

  Access to the TCP client sockets poll
  - TAsynchConnection.OnRead should rather use Write() and LogVerbose() methods of this
  TAsynchConnections class instead of using Clients

property Connection: TAsynchConnectionsObjArray read fConnection;

  Low-level unsafe direct access to the connection instances
  - ensure this property is used in a thread-safe manner, i.e. via
    Lock; try ... finally UnLock; end;

property ConnectionCount: integer read fConnectionCount;

  Low-level unsafe direct access to the connection count
  - ensure this property is used in a thread-safe manner, i.e. via
    Lock; try ... finally UnLock; end;

property LastOperationIdleSeconds: cardinal read fLastOperationIdleSeconds write fLastOperationIdleSeconds;

  Will execute TAsynchConnection.OnLastOperationIdle after an idle period
  - could be used to send heartbeats after read/write inactivity
  - equals 0 (i.e. disabled) by default

property Log: TSynLogClass read fLog;

  Access to the associated log class

property Options: TAsynchConnectionsOptions read fOptions write fOptions;

  Allow to customize low-level options for processing

TAsynchServer = class(TAsynchConnections)

  Implements a thread-pooled high-performance TCP server
  - will use a TAsynchConnection inherited class to maintain connection state for server process
constructor Create(const aPort: SockString; OnStart, OnStop: TNotifyThreadEvent;
aStreamClass: TAsynchConnectionClass; const ProcessName: SockString; aLog:
TSynLogClass; aOptions: TAsynchConnectionsOptions; aThreadPoolCount: integer=1);
reintroduce; virtual;
Run the TCP server, listening on a supplied IP port

destructor Destroy; override;
Shut down the server, releasing all associated threads and sockets

property Server: TCrtSocket read fServer;
Access to the TCP server socket

TAsynchClient = class(TAsynchConnections)
Implements thread-pooled high-performance TCP multiple clients
- e.g. to run some load stress tests with optimized resource use
- will use a TAsynchConnection inherited class to maintain connection state of each connected client

constructor Create(const aServer, aPort: SockString;
aClientsCount, aClientsTimeoutSecs: integer; OnStart, OnStop: TNotifyThreadEvent;
aStreamClass: TAsynchConnectionClass; const ProcessName: SockString; aLog:
TSynLogClass; aOptions: TAsynchConnectionsOptions; aThreadPoolCount: integer=1);
reintroduce; virtual;
Start the TCP client connections, connecting to the supplied IP server

property Port: SockString read fThreadClients.Port;
Server IP port

property Server: SockString read fThreadClients.Address;
Server IP address

EWebSockets = class(ESynException)
Exception raised when processing WebSockets

TWebSocketFrame = record
Stores a WebSockets frame
- see @http://tools.ietf.org/html/rfc6455 for reference

content: TWebSocketFramePayloads;
What is stored in the frame data, i.e. in payload field

opcode: TWebSocketFrameOpCode;
The interpretation of the frame data

payload: RawByteString;
The frame data itself
- is plain UTF-8 for focText kind of frame
- is raw binary for focBinary or any other frames

tix: cardinal;
Equals GetTickCount64 shr 10, as used for TWebSocketFrameList timeout
TWebSocketProtocol = class(TSynPersistent)

One instance implementing application-level WebSockets protocol
- shared by TWebSocketServer and TWebSocketClient classes
- once upgraded to WebSockets, a HTTP link could be used e.g. to transmit our proprietary
'synopsejson' or 'synopsebin' application content, as stated by this typical handshake:
GET /myservice HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JjHmbDL1EzLkh9Gb8xXn
Sec-WebSocket-Protocol: synopsejson
Sec-WebSocket-Version: 13
Origin: http://example.com

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMIYUKAGmm5OPpG2HaGw=
Sec-WebSocket-Protocol: synopsejson
- the TWebSocketProtocolJSON inherited class will implement
Sec-WebSocket-Protocol: synopsejson
- the TWebSocketProtocolBinary inherited class will implement
Sec-WebSocket-Protocol: synopsebin

constructor Create(const aName, aURI: RawUTF8); reintroduce;
Abstract constructor to initialize the protocol
- the protocol should be named, so that the client may be able to request for a given protocol
- if aURI is '', any URI would potentially upgrade to this protocol; you can specify an URI to limit
the protocol upgrade to a single resource

function Clone(const aClientURI: RawUTF8): TWebSocketProtocol; virtual; abstract;
Compute a new instance of the WebSockets protocol, with same parameters

procedure SetEncryptKey(aServer: boolean; const aKey: RawUTF8);
Set the fEncryption: IProtocol according to the supplied key
- any asymmetric algorithm need to know which side (client/server) to work on
- try TECHTTPSProtocol.FromKey(aKey) and fallback to TProtocolAES.Create(TAESCFB) using
SHA256Weak(aKey)

procedure SetEncryptKeyAES(const aKey; aKeySize: cardinal);
Set the fEncryption: IProtocol as TProtocolAES.Create(TAESCFB)

property Encrypted: boolean read GetEncrypted;
Returns TRUE if encryption is enabled during the transmission
- is currently only available for TWebSocketProtocolBinary

property Encryption: IProtocol read fEncryption;
Access low-level frame encryption

property FramesInBytes: QWord read fFramesInBytes;
How many (uncompressed) bytes have been received by this instance
property FramesInCount: integer read fFramesInCount;
   How many frames have been received by this instance

property FramesOutBytes: QWord read fFramesOutBytes;
   How many (uncompressed) bytes have been sent by this instance

property FramesOutCount: integer read fFramesOutCount;
   How many frames have been sent by this instance

property LastError: string read fLastError;
   The last error message, during frame processing

property Name: RawUTF8 read fName;
   The Sec-WebSocket-Protocol application name currently involved
      - e.g. 'synopsejson', 'synopsebin' or 'synopsebinary'

property OnBeforeIncomingFrame: TOnWebSocketProtocolIncomingFrame read fOnBeforeIncomingFrame
   write fOnBeforeIncomingFrame;
   Allow low-level interception before ProcessIncomingFrame is done

property RemoteIP: SockString read GetRemoteIP;
   The associated 'Remote-IP' HTTP header value
      - returns "" if self=nil or RemoteLocalhost=true

property UpgradeURI: RawUTF8 read fUpgradeURI;
   The URI on which this protocol has been upgraded

property URI: RawUTF8 read fURI;
   The optional URI on which this protocol would be enabled
      - leave to "" if any URI should match

TWebSocketProtocolChat = class(TWebSocketProtocol)
   Simple chatting protocol, allowing to receive and send WebSocket frames
      - you can use this protocol to implement simple asynchronous communication with events
         expecting no answers, e.g. with AJAX applications
      - see TWebSocketProtocolRest for bi-directional events expecting answers

constructor Create(const aName,aURI: RawUTF8; const aOnIncomingFrame:
      TOnWebSocketProtocolChatIncomingFrame); overload;
   Initialize the chat protocol with an incoming frame callback

function Clone(const aClientURI: RawUTF8): TWebSocketProtocol; override;
   Compute a new instance of the WebSockets protocol, with same parameters

function SendFrame(Sender: THttpServerResp; const Frame: TWebSocketFrame): boolean;
   On the server side, allows to send a message over the wire to a specified client connection
      - a temporary copy of the Frame content will be made for safety

function SendFrameJson(Sender: THttpServerResp; var JSON: RawUTF8): boolean;
   On the server side, allows to send a JSON message over the wire to a specified client connection
      - the supplied JSON content is supplied as "var", since it may be modified during execution, e.g.
         XORed for frame masking
property OnIncomingFrame: TOnWebSocketProtocolChatIncomingFrame read fOnIncomingFrame write fOnIncomingFrame;

You can assign an event to this property to be notified of incoming messages

TWebSocketProtocolRest = class(TWebSocketProtocol)

Handle a REST application-level bi-directional WebSockets protocol
- will emulate a bi-directional REST process, using THttpServerRequest to store and handle the request parameters: clients would be able to send regular REST requests to the server, but the server could use the same communication channel to push REST requests to the client
- a local THttpServerRequest will be used on both client and server sides, to store REST parameters and compute the corresponding WebSockets frames

TWebSocketProtocolJSON = class(TWebSocketProtocolRest)

Handle a REST application-level WebSockets protocol using JSON for transmission
- could be used e.g. for AJAX or non Delphi remote access
- this class will implement then following application-level protocol:
  Sec-WebSocket-Protocol: synopsejson

constructor Create(const aURI: RawUTF8); reintroduce;

Initialize the WebSockets JSON protocol
- if aURI is ", any URI would potentially upgrade to this protocol; you can specify an URI to limit the protocol upgrade to a single resource

function Clone(const aClientURI: RawUTF8): TWebSocketProtocol; override;

Compute a new instance of the WebSockets protocol, with same parameters

TWebSocketProtocolBinary = class(TWebSocketProtocolRest)

Handle a REST application-level WebSockets protocol using compressed and optionally AES-CFB encrypted binary
- this class will implement then following application-level protocol:
  Sec-WebSocket-Protocol: synopsebin
  or fallback to the previous subprotocol
  Sec-WebSocket-Protocol: synopsebinary

- 'synopsebin' will expect requests sequenced as 'r000001','r000002',... headers matching 'a000001','a000002',... instead of 'request'/'answer'

constructor Create(const aURI: RawUTF8; aServer: boolean; const aKey: RawUTF8; aCompressed: boolean=true); reintroduce; overload;

Initialize the WebSockets binary protocol from a textual key
- if aURI is ", any URI would potentially upgrade to this protocol; you can specify an URI to limit the protocol upgrade to a single resource
- will create a TProtocolAES or TECDHEProtocol instance, corresponding to the supplied aKey and aServer values, to secure the transmission using a symmetric or assymetric algorithm
- SynLZ compression is enabled by default, unless aCompressed is false
constructor Create(const aURI: RawUTF8; const aKey; aKeySize: cardinal; aCompressed: boolean=true); reintroduce; overload;

*Initialize the WebSockets binary protocol with a symmetric AES key*
- if aURI is '', any URI would potentially upgrade to this protocol; you can specify an URI to limit the protocol upgrade to a single resource
- if aKeySize is 128, 192 or 256, TProtocolAES (i.e. AES-CFB encryption) will be used to secure the transmission
- SynLZ compression is enabled by default, unless aCompressed is false

constructor Create(const aURI: RawUTF8; aCompressed: boolean=true); reintroduce; overload; virtual;

*Initialize the WebSockets binary protocol with no encryption*
- if aURI is '', any URI would potentially upgrade to this protocol; you can specify an URI to limit the protocol upgrade to a single resource
- SynLZ compression is enabled by default, unless aCompressed is false

function Clone(const aClientURI: RawUTF8): TWebSocketProtocol; override;

*Compute a new instance of the WebSockets protocol, with same parameters*

property Compressed: boolean read fCompressed write fCompressed;

*Defines if SynLZ compression is enabled during the transmission*
- is set to TRUE by default

property FramesInBytesSocket: QWord read fFramesInBytesSocket;

*How many bytes have been received by this instance from the wire*

property FramesInCompression: integer read GetFramesInCompression;

*Compression ratio of frames received by this instance*

property FramesOutBytesSocket: QWord read fFramesOutBytesSocket;

*How many bytes have been sent by this instance to the wire*

property FramesOutCompression: integer read GetFramesOutCompression;

*Compression ratio of frames Sent by this instance*

TWebSocketProtocolList = class(TSynPersistentLock)

*Used to maintain a list of websocket protocols (for the server side)*

destructor Destroy; override;

*Finalize the list storage*

function Add(aProtocol: TWebSocketProtocol): boolean;

*Add a protocol to the internal list*
- returns TRUE on success
- if this protocol is already existing for this given name and URI, returns FALSE: it is up to the caller to release aProtocol if needed
function AddOnce(aProtocol: TWebSocketProtocol): boolean;
    Add once a protocol to the internal list
    - if this protocol is already existing for this given name and URI, any previous one will be
      released - so it may be confusing on a running server
    - returns TRUE if the protocol was added for the first time, or FALSE if the protocol has been
      replaced or is invalid (e.g. aProtocol=nil)

function CloneByName(const aProtocolName, aClientURI: RawUTF8): TWebSocketProtocol;
    Create a new protocol instance, from the internal list

function CloneByURI(const aClientURI: RawUTF8): TWebSocketProtocol;
    Create a new protocol instance, from the internal list

function Count: integer;
    How many protocols are stored

function Remove(const aProtocolName, aURI: RawUTF8): boolean;
    Erase a protocol from the internal list, specified by its name

TWebSocketFrameList = class(TSynPersistentLock)
    Used to manage a thread-safe list of WebSockets frames

    Count: integer;
        Current number of WebSocket frames in the list

    List: TWebSocketFrameDynArray;
        Low-level access to the WebSocket frames list

    constructor Create(timeoutsec: integer); reintroduce;
        Initialize the list

function AnswerToIgnore(incr: integer=0): integer;
    How many 'answer' frames are to be ignored
    - this method is thread-safe

function Pop(protocol: TWebSocketProtocol; const head: RawUTF8; out frame: TWebSocketFrame): boolean;
    Retrieve a WebSocket frame from the list, oldest first
    - you should specify a frame type to search for, according to the specified WebSockets protocol
    - this method is thread-safe

procedure Push(const frame: TWebSocketFrame);
    Add a WebSocket frame in the list
    - this method is thread-safe

procedure PushVoidFrame(opcode: TWebSocketFrameOpCode);
    Add a void WebSocket frame in the list
    - this method is thread-safe

TWebSocketProcessSettings = object(TObject)
    Parameters to be used for WebSockets process
CallbackAcquireTimeOutMS: cardinal;

How many milliseconds the callback notification should wait acquiring the connection before failing
- default is 5000, i.e. 5 seconds

CallbackAnswerTimeOutMS: cardinal;

How many milliseconds the callback notification should wait for the client to return its answer
- default is 30000, i.e. 30 seconds

DisconnectAfterInvalidHeartbeatCount: cardinal;

Will close the connection after a given number of invalid Heartbeat sent
- when a Heartbeat is failed to be transmitted, the class will start counting how many ping/pong did fail: when this property value is reached, it will release and close the connection
- default value is 5

HeartbeatDelay: cardinal;

Time in milliseconds between each focPing commands sent to the other end
- default is 0, i.e. no automatic ping sending on client side, and 20000, i.e. 20 seconds, on server side

LogDetails: set of (logHeartbeat, logTextFrameContent, logBinaryFrameContent);

By default, contains [] to minimize the logged information
- set logHeartbeat if you want the ping/pong frames to be logged
- set logTextFrameContent if you want the text frame content to be logged
- set logBinaryFrameContent if you want the binary frame content to be logged
- used only if WebSocketLog global variable is set to a TSynLog class

LoopDelay: cardinal;

Maximum period time in milli seconds when ProcessLoop thread will stay idle before checking for the next pending requests
- default is 500 ms, but you may put a lower value, if you expects e.g. REST commands or NotifyCallback(wscNonBlockWithoutAnswer) to be processed with a lower delay

OnClientConnected: TNotifyEvent;

Callback run when a WebSockets client is just connected
- triggered by TWebSocketProcess.ProcessStart

OnClientDisconnected: TNotifyEvent;

Callback run when a WebSockets client is just disconnected
- triggered by TWebSocketProcess.ProcessStop

SendDelay: cardinal;

Ms between sending - allow to gather output frames
- GetTickCount resolution is around 16ms under Windows, so default 10ms seems fine for a cross-platform similar behavior

procedure SetDefaults;

Will set the default values

procedure SetFullLog;

Will set LogDetails to its highest level of verbosity
- used only if WebSocketLog global variable is set
TWebSocketProcess = class(TSynPersistent)

Abstract WebSockets process, used on both client or server sides
- CanGetFrame/ReceiveBytes/SendBytes abstract methods should be overriden with actual communication, and fState and ProcessStart/ProcessStop should be updated from the actual processing thread (e.g. as in TWebCrtSocketProcess)

constructor Create(aProtocol: TWebSocketProtocol; aOwnerConnection: THttpServerConnectionID; aOwnerThread: TSynThread; const aSettings: TWebSocketProcessSettings; const aProcessName: RawUTF8); reintroduce;

Initialize the WebSockets process on a given connection
- the supplied TWebSocketProtocol will be owned by this instance
- other parameters should reflect the client or server expectations

destructor Destroy; override;

Finalize the context
- if needed, will notify the other end with a focConnectionClose frame
- will release the TWebSocketProtocol associated instance

function CanGetFrame(TimeOut: cardinal; ErrorWithoutException: PInteger): boolean;
virtual; abstract;

Abstract low-level method to check if there is some pending input data in the input Socket ready for GetFrame/ReceiveBytes
- is defined separated to allow multi-thread pooling

function GetFrame(out Frame: TWebSocketFrame; ErrorWithoutException: PInteger): boolean;

Blocking process incoming WebSockets framing protocol
- CanGetFrame should have been called and returned true before
- will call overriden ReceiveBytes() for the actual communication

function NotifyCallback(aRequest: THttpServerRequest; aMode: TWebSocketProcessNotifyCallback): cardinal; virtual;

Will push a request or notification to the other end of the connection
- the caller should set the aRequest with the outgoing parameters, and optionally receive a response from the other end
- the request may be sent in blocking or non blocking mode
- returns the HTTP Status code (e.g. HTTP_SUCCESS=200 for success)

function ReceiveBytes(P: PAnsiChar; count: integer): integer; virtual; abstract;

Abstract low-level method to retrieve pending input data
- should return the number of bytes (<=count) received and written to P
- is defined separated to allow multi-thread pooling

function RemoteIP: SockString;

The associated 'Remote-IP' HTTP header value
- returns "" if Protocol=nil or Protocol.RemoteLocalhost=true

function SendBytes(P: pointer; Len: integer): boolean; virtual; abstract;

Abstract low-level method to send pending output data
- returns false on any error, try on success
- is defined separated to allow multi-thread pooling
function SendFrame(var Frame: TWebSocketFrame): boolean;

  Process outgoing WebSockets framing protocol \to be overridden
  \: will call overridden SendBytes() for the actual communication
  \: use Outgoing.Push() to send frames asynchronously

function Settings: PWebSocketProcessSettings;
  \: The settings currently used during the WebSockets process
  \: defined as a pointer so that you may be able to change the values

function State: TWebSocketProcessState;
  \: Returns the current state of the underlying connection

property Incoming: TWebSocketFrameList read fIncoming;
  \: Direct access to the low-level incoming frame stack

property InvalidPingSendCount: cardinal read fInvalidPingSendCount;
  \: How many invalid heartbeat frames have been sent
  \: a non 0 value indicates a connection problem

property NoConnectionCloseAtDestroy: boolean read fNoConnectionCloseAtDestroy write fNoConnectionCloseAtDestroy;
  \: May be set to TRUE before Destroy to force raw socket disconnection

property Outgoing: TWebSocketFrameList read fOutgoing;
  \: Direct access to the low-level outgoing frame stack
  \: call Outgoing.Push() to send frames asynchronously, with optional jumboframe gathering (if supported by the protocol)

property OwnerConnection: THttpServerConnectionID read fOwnerConnection;
  \: The associated low-level WebSocket connection opaque identifier

property OwnerThread: TSynThread read fOwnerThread;
  \: The associated low-level processing thread

property ProcessCount: integer read fProcessCount;
  \: How many frames are currently processed by this connection

property ProcessName: RawUTF8 read fProcessName write fProcessName;
  \: The associated process name

property Protocol: TWebSocketProtocol read fProtocol;
  \: The Sec-WebSocket-Protocol application protocol currently involved
  \: TWebSocketProtocolJSON or TWebSocketProtocolBinary in the mORMot context
  \: could be nil if the connection is in standard HTTP/1.1 mode

TWebCrtSocketProcess = class(TWebSocketProcess)
  \: TCrtSocket-based WebSockets process, used on both client or server sides
  \: will use the socket in blocking mode, so expects its own processing thread
constructor Create(aSocket: TCrtSocket; aProtocol: TWebSocketProtocol;
aOwnerConnection: THttpServerConnectionID; aOwnerThread: TSynThread; const
aSettings: TWebSocketProcessSettings; const aProcessName: RawUTF8); reintroduce;
virtual;

Initialize the WebSockets process on a given TCrtSocket connection
- the supplied TWebSocketProtocol will be owned by this instance
- other parameters should reflect the client or server expectations

function CanGetFrame(TimeOut: cardinal; ErrorWithoutException: PInteger): boolean; override;

First step of the low level incoming WebSockets framing protocol over TCrtSocket
- in practice, just call fSocket.SockInPending to check for pending data

function ReceiveBytes(P: PAnsiChar; count: integer): integer; override;

Low level receive incoming WebSockets frame data over TCrtSocket
- in practice, just call fSocket.SockInRead to check for pending data

function SendBytes(P: pointer; Len: integer): boolean; override;

Low level receive incoming WebSockets frame data over TCrtSocket
- in practice, just call fSocket.TrySndLow to send pending data

property Socket: TCrtSocket read fSocket;

The associated communication socket
- on the server side, is a THttpServerSocket
- access to this instance is protected by Safe.Lock/Unlock

TWebSocketProcessServer = class(TWebCrtSocketProcess)
Implements WebSockets process as used on server side

TWebSocketServerResp = class(THttpServerResp)
An enhanced input/output structure used for HTTP and WebSockets requests
- this class will contain additional parameters used to maintain the WebSockets execution context
in overridden TWebSocketServer.Process method

constructor Create(aServerSock: THttpServerSocket; aServer: THttpServer);
override;

Initialize the context, associated to a HTTP/WebSockets server instance

function NotifyCallback(Ctxt: THttpServerRequest; aMode:
TWebSocketProcessNotifyCallback): cardinal; virtual;

Push a notification to the client

function WebSocketProtocol: TWebSocketProtocol;

The Sec-WebSockets-Protocol application protocol currently involved
- TWebSocketProtocolJSON or TWebSocketProtocolBinary in the mORMot context
- could be nil if the connection is in standard HTTP/1.1 mode

property WebSocketProcess: TWebSocketProcessServer read fProcess;

Low-level WebSocket protocol processing instance
TWebSocketServer = class(THttpServer)

Main HTTP/WebSockets server Thread using the standard Sockets API (e.g. WinSock)
- once upgraded to WebSockets from the client, this class is able to serve any Sec-WebSocket-Protocol application content

constructor Create(const aPort: SockString; OnStart, OnStop: TNotifyThreadEvent;
const ProcessName: SockString; ServerThreadPoolCount: integer = 2;
KeepAliveTimeOut: integer = 30000; HeadersNotFiltered: boolean = false;
CreateSuspended: boolean = false); override;

Create a Server Thread, binded and listening on a port
- this constructor will raise a EHttpServer exception if binding failed
- expects the port to be specified as string, e.g. '1234'; you can optionally specify a server address to bind to, e.g. '1.2.3.4:1234'
- due to the way how WebSockets works, one thread will be created for any incoming connection
- note that this constructor will not register any protocol, so is useless until you execute Protocols.Add()
- in the current implementation, the ServerThreadPoolCount parameter will use two threads by default to handle shortliving HTTP/1.0 "connection: close" requests, and one thread will be maintained per keep-alive/websockets client
- by design, the KeepAliveTimeOut value is ignored with this server once it has been upgraded to WebSockets

destructor Destroy; override;

Close the server

function IsActiveWebSocket(ConnectionID: THttpServerConnectionID):
TWebSocketServerResp;
Give access to the underlying connection from its ID
- also identifies an incoming THttpServerResp as a valid TWebSocketServerResp

function IsActiveWebSocketThread(ConnectionThread: TSynThread):
TWebSocketServerResp;
Give access to the underlying connection from its connection thread
- also identifies an incoming THttpServerResp as a valid TWebSocketServerResp

function Settings: PWebSocketProcessSettings;
The settings to be used for WebSockets process
- note that those parameters won't be propagated to existing connections
- defined as a pointer so that you may be able to change the values

function WebSocketConnections: integer;
How many WebSockets connections are currently maintained

procedure WebSocketBroadcast(const aFrame: TWebSocketFrame); overload;
Will send a given frame to all connected clients
- expect aFrame.opcode to be either focText or focBinary
- will call TWebSocketProcess.Outgoing.Push for asynchronous sending
procedure WebSocketBroadcast(const aFrame: TWebSocketFrame; const aClientsConnectionID: THttpServerConnectionIDDynArray); overload;

Will send a given frame to clients matching the supplied connection IDs
- expect aFrame.opcode to be either focText or focBinary
- will call TWebSocketProcess.Outgoing.Push for asynchronous sending

property WebSocketProtocols: TWebSocketProtocolList read fProtocols;
Access to the protocol list handled by this server

TWebSocketServerRest = class(TWebSocketServer)

Main HTTP/WebSockets server Thread using the standard Sockets API (e.g. WinSock)
- once upgraded to WebSockets from the client, this class is able to serve our proprietary
Sec-WebSocket-Protocol: 'synopsejson' or 'synopsebin' application content, managing regular
REST client-side requests and also server-side push notifications
- once in 'synopse*' mode, the Request() method will be triggered from any incoming REST
request from the client, and the OnCallback event will be available to push a request from the
server to the client

constructor Create(const aPort: SockString; OnStart, OnStop: TNotifyThreadEvent; const aProcessName, aWebSocketsURI, aWebSocketsEncryptionKey: RawUTF8; aWebSocketsAJAX: boolean=false); reintroduce; overload;

Create a Server Thread, binded and listening on a port, with our 'synopsebin' and optionally
'synopsejson' modes
- if aWebSocketsURI is '', any URI would potentially upgrade; you can specify an URI to limit the
protocol upgrade to a single resource
- TWebSocketProtocolBinary will always be registered by this constructor
- if the encryption key text is not '', TWebSocketProtocolBinary will use AES-CFB 256 bits
encryption
- if aWebSocketsAJAX is TRUE, it will also register TWebSocketProtocolJSON so that AJAX
applications would be able to connect to this server
- warning: WaitStarted should be called after Create() to check for actual port binding in the
background thread

function Callback(Ctxt: THttpServerRequest; aNonBlocking: boolean): cardinal; override;

Server can send a request back to the client, when the connection has been upgraded to
WebSocket
- InURL/InMethod/InContent properties are input parameters (InContentType is ignored)
- OutContent/OutContentType/OutCustomHeader are output parameters
- CallingThread should be set to the client's Ctxt.CallingThread value, so that the method could
know which connection is to be used - it will return STATUS_NOTFOUND (404) if the
connection is unknown
- result of the function is the HTTP error code (200 if OK, e.g.)
procedure WebSocketsEnable(const aWebSocketsURI, aWebSocketsEncryptionKey: RawUTF8; aWebSocketsAJAX: boolean=false; aWebSocketsCompressed: boolean=true);

Defines the WebSockets protocols to be used for this Server
- i.e. 'synopsebin' and optionally 'synopsejson' modes
- if aWebSocketsURI is '', any URI would potentially upgrade; you can specify an URI to limit the protocol upgrade to a single resource
- TWebSocketProtocolBinary will always be registered by this constructor
- if the encryption key text is not '', TWebSocketProtocolBinary will use AES-CFB 256 bits encryption
- if aWebSocketsAJAX is TRUE, it will also register TWebSocketProtocolJSON so that AJAX applications would be able to connect to this server

TWebSocketProcessClient = class(TWebCrtSocketProcess)

Implements WebSockets process as used on client side

constructor Create(aSender: THttpClientWebSockets; aProtocol: TWebSocketProtocol; const aProcessName: RawUTF8); reintroduce; virtual;

Initializes the client process for a given THttpClientWebSockets

destructor Destroy; override;

Finalizes the process

TWebSocketProcessClientThread = class(TSynThread)

WebSockets processing thread used on client side
- will handle any incoming callback

THttpClientWebSockets = class(THttpClientSocket)

Socket API based REST and HTTP/1.1 client, able to upgrade to WebSockets
- will implement regular HTTP/1.1 until WebSocketsUpgrade() is called

constructor Create(aTimeOut: PtrInt=10000); override;

Common initialization of all constructors
- this overridden method will set the UserAgent with some default value

destructor Destroy; override;

Finalizes the connection

function Request(const url, method: SockString; KeepAlive: cardinal; const header, Data, DataType: SockString; retry: boolean): integer; override;

Process low-level REST request, either on HTTP/1.1 or via WebSockets
- after WebSocketsUpgrade() call, will use WebSockets for the communication

function Settings: PWebSocketProcessSettings;

The settings to be used for WebSockets process
- note that those parameters won't be propagated to existing connections
- defined as a pointer so that you may be able to change the values
class function WebSocketsConnect(const aHost, aPort: SockString; aProtocol: TWebSocketProtocol; aLog: TSynLogClass=nil; const aLogContext: RawUTF8=''; const aURI: RawUTF8=''; const aCustomHeaders: RawUTF8=''): THtClientWebSockets;

Low-level initialization of a client WebSockets connection
- calls Open() then WebSocketsUpgrade() for a given protocol
- with proper error interception and optional logging, returning nil

function WebSocketsUpgrade(const aWebSocketsURI, aWebSocketsEncryptionKey: RawUTF8; aWebSocketsAJAX: boolean=false; aWebSocketsCompression: boolean=true; aProtocol: TWebSocketProtocol=nil; const aCustomHeaders: RawUTF8=''): RawUTF8;

Upgrade the HTTP client connection to a specified WebSockets protocol
- i.e. 'synopsebin' and optionally 'synopsejson' modes
- you may specify an URI to as expected by the server for upgrade
- if aWebSocketsAJAX equals default FALSE, it will register the
TWebSocketProtocolBinaryprotocol, with AES-CFB 256 bits encryption if the encryption key text is not " and optional SynLZ compression
- if aWebSocketsAJAX is TRUE, it will register the slower and less secure
TWebSocketProtocolJSON (to be used for AJAX debugging/test purposes only) and
aWebSocketsEncryptionKey/aWebSocketsCompression parameters won't be used
- alternatively, you can specify your own custom TWebSocketProtocol instance (owned by this
method and immediately released on error)
- will return " on success, or an error message on failure

property OnBeforeIncomingFrame: TOnWebSocketProtocolIncomingFrame read fOnBeforeIncomingFrame write fOnBeforeIncomingFrame;

Allow low-level interception before TWebSocketProcessClient.ProcessIncomingFrame is executed

property OnCallbackRequestProcess: TOnHttpServerRequest read fOnCallbackRequestProcess write fOnCallbackRequestProcess;

This event handler will be executed for any incoming push notification

property OnWebSocketsClosed: TNotifyEvent read fOnWebSocketsClosed write fOnWebSocketsClosed;

Event handler triggered when the WebSocket link is destroyed
- may happen e.g. after graceful close from the server side, or after
DisconnectAfterInvalidHeartbeatCount is reached

property WebSockets: TWebSocketProcessClient read fProcess;

The current WebSockets processing class
- equals nil for plain HTTP/1.1 mode
- points to the current WebSockets process instance, after a successful WebSocketsUpgrade() call, so that you could use e.g. WebSockets.Protocol to retrieve the protocol currently used

Types implemented in the SynBidirSock unit

PWebSocketFrame = ^TWebSocketFrame;

Points to a WebSockets frame

PWebSocketProcessSettings = ^TWebSocketProcessSettings;

Points to parameters to be used for WebSockets process
- using a pointer/reference type will allow in-place modification of any TWebSocketProcess.Settings,
TWebSocketServer.Settings or THtClientWebSockets.Settings property

TAsynchConnectionClass = class of TAsynchConnection;
Meta-class of one TAsynchConnections connection

**TAsynchConnectionHandle** = `type` integer;

32-bit integer value used to identify an asynchronous connection
- will start from 1, and increase during the TAsynchConnections live-time

**TAsynchConnectionObjArray** = `array of` TAsynchConnection;

Used to store a dynamic array of TAsynchConnection

**TAsynchConnectionsOptions** = `set of` (acoOnErrorContinue, acoOnAcceptFailureStop, acoNoLogRead, acoNoLogWrite, acoVerboseLog, acoLastOperationNoRead, acoLastOperationNoWrite);

Low-level options for TAsynchConnections processing
- TAsynchConnections.Sockets.OnError will shutdown the connection on any error, unless acoOnErrorContinue is defined
- acoOnAcceptFailureStop will let failed Accept() finalize the process
- acoNoLogRead and acoNoLogWrite could reduce the log verbosity
- acoVerboseLog will log transmitted frames content, for debugging purposes
- acoLastOperationNoRead and acoLastOperationNoWrite could be used to avoid TAsynchConnection.fLastOperation reset at read or write

**THttpRequestCacheDynArray** = `array of` THttpRequestCache;

In-memory storage of all THttpRequestCached entries

**TOnWebSocketProtocolChatIncomingFrame** = `procedure` (Sender: THttpServerResp; `const` Frame: TWebSocketFrame) `of` object;

Callback event triggered by TWebSocketProtocolChat for any incoming message
- a first call with frame.opcode=focContinuation will take place when the connection will be upgraded to WebSockets
- then any incoming focText/focBinary events will trigger this callback
- eventually, a focConnectionClose will notify the connection ending

**TOnWebSocketProtocolIncomingFrame** = `function` (Sender: TWebSocketProcess; `var` Frame: TWebSocketFrame): `boolean` `of` object;

Callback event triggered by TWebSocketProtocol for any incoming message
- called before TWebSocketProtocol.ProcessIncomingFrame for incoming focText/focBinary frames
- should return true if the frame has been handled, or false if the regular processing should take place

**TWebSocketFrameDynArray** = `array of` TWebSocketFrame;

A dynamic list of WebSockets frames

**TWebSocketFrameOpCode** = (focContinuation, focText, focBinary, focReserved3, focReserved4, focReserved5, focReserved6, focReserved7, focConnectionClose, focPing, focPong, focReserved8, focReserved9, focReservedE, focReservedF);

Defines the interpretation of the WebSockets frame data
- match order expected by the WebSockets RFC

**TWebSocketFrameOpCodes** = `set of` TWebSocketFrameOpCode;

Set of WebSockets frame interpretation

**TWebSocketFramePayload** = (fopAlreadyCompressed);

Define one attribute of a WebSockets frame data

**TWebSocketFramePayloads** = `set of` TWebSocketFramePayload;
Define the attributes of a WebSockets frame data

TWebSocketProcessClientThreadState = ( sCreate, sRun, sFinished, sClosed );

The current state of the client side processing thread

TWebSocketProcessNotifyCallback = ( wscBlockWithAnswer, wscBlockWithoutAnswer, wscNonBlockWithoutAnswer );

Indicates how TWebSocketProcess.NotifyCallback() will work

TWebSocketProcessOne = ( wspNone, wspPing, wspDone, wspAnswer, wspError, wspClosed );

Indicates which kind of process did occur in the main WebSockets loop

TWebSocketProcessState = ( wpsCreate, wpsRun, wpsClose, wpsDestroy );

The current state of the WebSockets process

TWebSocketProtocolClass = class of TWebSocketProtocol;

Used to store the class of a TWebSocketProtocol type

### Functions or procedures implemented in the `SynBidirSock` unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FrameInit</td>
<td>Low-level initialization of a TWebSocketFrame for proper REST content</td>
<td>702</td>
</tr>
<tr>
<td>PurgeHeaders</td>
<td>Will remove most usual HTTP headers which are to be recomputed on sending</td>
<td>702</td>
</tr>
<tr>
<td>ToText</td>
<td>Used to return the text corresponding to a specified WebSockets frame data</td>
<td>702</td>
</tr>
<tr>
<td>ToText</td>
<td>Used to return the text corresponding to a specified WebSockets sending mode</td>
<td>702</td>
</tr>
</tbody>
</table>

**procedure** FrameInit(opcode: TWebSocketFrameOpCode; const Content, ContentType: RawByteString; out frame: TWebSocketFrame);

Low-level initialization of a TWebSocketFrame for proper REST content

**function** PurgeHeaders(P: PUTF8Char): RawUTF8;

Will remove most usual HTTP headers which are to be recomputed on sending

**function** ToText(mode: TWebSocketProcessNotifyCallback): PShortString; overload;

Used to return the text corresponding to a specified WebSockets sending mode

**function** ToText(opcode: TWebSocketFrameOpCode): PShortString; overload;

Used to return the text corresponding to a specified WebSockets frame data

### Variables implemented in the `SynBidirSock` unit
WebSocketLog: TSynLogClass;

If set, will log all WebSockets raw information
- see also TWebSocketProcessSettings.LogDetails and TWebSocketProcessSettings.SetFullLog to setup even more verbose information, e.g. by setting HttpServerFullWebSocketsLog and HttpClientFullWebSocketsLog global variables to true (as defined in mORMotHttpServer/mORMotHttpClient)

WebSocketsBinarySynLzThreshold: integer = 450;

Number of bytes above which SynLZ compression may be done
- when working with TWebSocketProtocolBinary
- it is useless to compress smaller frames, which fits in network MTU

WebSocketsIVReplayAttackCheck: TAESIVReplayAttackCheck = repNoCheck;

How replay attacks will be handled in TWebSocketProtocolBinary encryption
- you may set this global value to repCheckedIfAvailable if you are really paranoid (but resulting security may be lower, since the IV is somewhat more predictable than plain random)

WebSocketsMaxFrameMB: cardinal = 256;

The allowed maximum size, in MB, of a WebSockets frame
27.4. SynBigTable.pas unit

Purpose: Class used to store huge amount of data with fast retrieval
- licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the SynBigTable unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| SynCommons    | Common functions used by most Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 717  |
| SynTable      | Filter/database/cache/buffer/security/search/multithread/OS features  
- as a complement to SynCommons, which tended to increase too much  
- licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1721 |
| SynTests      | Unit test functions used by Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1832 |

Objects implemented in the SynBigTable unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSynBigTable</td>
<td>A class to store huge amount of data, just specified by an integer ID</td>
<td>705</td>
</tr>
<tr>
<td>TSynBigTableMetaData</td>
<td>A class to store huge data (like files content), with metadata fields associated with every record</td>
<td>712</td>
</tr>
<tr>
<td>TSynBigTableRecord</td>
<td>A class to store huge amount of data, with fields in every record</td>
<td>714</td>
</tr>
<tr>
<td>TSynBigTableString</td>
<td>A class to store huge amount of data, just specified by a string ID</td>
<td>708</td>
</tr>
<tr>
<td>TSynBigTableTable</td>
<td>An abstract class, associating a TSynTable to a Big Table</td>
<td>710</td>
</tr>
<tr>
<td>TTestBigTable</td>
<td>Unitary testing of the SynBigTable unit</td>
<td>715</td>
</tr>
</tbody>
</table>
TSynBigTable = class(TObject)

A class to store huge amount of data, just specified by an integer ID
- data is stored in an unique file
- retrieval is very fast (can't be faster IMHO)
- data is appended at the end of this file at adding (but use a caching mechanism for immediate adding)
- use a temporary in memory adding, till the UpdateToFile method is called
- data items can be deleted
- file can be packed using the Pack method in order to retrieve free space from deleted entries
  (sounds like a VACUUM command, but faster)
- total size of file has no limit (but your hard disk, of course)
- limit of one data block depends on RAM (RawByteString is used as storage for data block)
- before Delphi 2007, much faster when using FastMM4 memory manager
- after profiling, most of the time is spent in the Windows kernel, waiting from hard disk write of raw data; in all cases, this class is much faster than any SQL engine storing BLOB, and than plain Win32 files
- ACID behavior can be enforced by calling UpdateToFile(true)

constructor Create(const aFileName: TFileName; FileOpenMode: Cardinal = 0);
reintroduce;

  Initialize the database engine with a supplied filename
  - InternalCacheSize can be used to customize the internal cache count (set to 0 to disable caching; 128 is a good value, bigger makes no difference)
  - you can specify a custom file open mode attributes on request, like fmShareDenyNone (not set by default, for safety reason)

destructor Destroy; override;

  Finalize memory, and save all content

function Add(const aData: RawByteString; ForcedID: integer=0; PhysicalIndex: PInteger=nil; OldPhysicalIndex: integer=-1): integer; virtual;

  Add a data to the database
  - return the unique ID created to identify this data
  - you can force a specified ID number, by setting a non null value to ForcedID (in this case, it MUST be added in increasing order)
  - return 0 on error, otherwise the created Physical Index can be stored
  - OldPhysicalIndex is to be used in case of constraint check (for TSynBigTableRecord and TSynBigTableMetaData)

function AddFile(const aFileName: TFileName): integer;

  Add a file content to the database
  - return the unique ID created to identify this data
  - return 0 on error (e.g. specified file doesn't exist)

function Delete(aID: integer; PhysicalIndex: PInteger=nil): boolean; virtual;

  Delete an ID
function Get(aIDFirst, aIDLast: integer; out aData: TRawByteStringDynArray): boolean; overload;
  Retrieve a list of content, from a supplied ID range (including boundaries)
  - return TRUE if all were found, FALSE if some ID were not existing (or deleted)
  - return the data into aData[], or " if one particular ID was not existing (or deleted); after call,
    length(aData)=aIDLast-aIDFirst+1

function Get(aID: integer; out aData: RawByteString): boolean; overload;
  Retrieve a content, from a supplied ID
  - return TRUE if found, FALSE if ID was not existing (or deleted)
  - return the data into aData, or " if ID was not existing (or deleted)

function GetAllIDs(var IDs: TIntegerDynArray; Order: TSynBigTableIterationOrder=ioPhysical): integer; virtual;
  Fast retrieval of all IDs
  - returned in physical or increasing ID value order
  - returns the number of IDs stored in the integer array

function GetAllPhysicalIndexes(var Indexes: TIntegerDynArray): integer;
  Fast retrieval of all used items physical indexes
  - returned in physical order
  - returns the number of Indexes stored in the integer array

function GetAsStream(aID: integer): TStream;
  Retrieve a content, from a supplied ID, into a TStream
  - this method is faster than Get() into a RawByteString, because the data is not moved from
    memory but mapped into a TCustomMemoryStream
  - if the ID is not correct, returns nil
  - if the ID is correct, returns a TStream instance, able to access to the associated content
  - in most cases, this TStream is just a wrapper around the memory mapped buffer in memory
  - the TStream must be consumed immediately, before any Pack or UpdateToFile method calls
  - the caller must Free the returned TStream instance
  - if the data is not already memory mapped (i.e. for files >= 2 GB) a custom
    TSynMemoryStreamMapped is used to access the data from disk

function GetLength(aID: integer): integer;
  Retrieve the length of a content, from a supplied ID
  - return -1 if the ID was not found, or the length (in bytes) of this ID content

function GetPart(aID: integer; Offset, Len: Integer; out aData: RawByteString): boolean;
  Retrieve a part of a file content
  - faster than Open/Seek/Read methods, which loads the whole content in memory before Seek
    and Read
  - only the needed part of data is copied into aData
function GetPointer(aID: Integer; var aTempData: RawByteString; DataLen: PInteger=nil): pointer;

*Retrieve a content, from a supplied ID, into a pointer*
- returns nil on error
- returns a pointer to the data on success, directly from the memory mapped file on most cases;
- if the data is not in a memory mapped buffer (i.e. for files >= 2 GB) the aTempData variable is used to read the data from disk
- in case of success, if DataLen is not nil, it will be filled with the corresponding data length
- this method is therefore much faster than Get()

function GetPointerFromPhysicalIndex(aPhysicalIndex: integer; var aTempData: RawByteString): pointer;

*Retrieve a content, from a supplied ID, into a pointer*
- returns nil on error
- returns a pointer to the data on success, directly from the memory mapped file on most cases;
- if the data is not in a memory mapped buffer (i.e. for files >= 2 GB) the aTempData variable is used to read the data from disk
- this method is not thread-safe (but is therefore faster)

function Update(aID: Integer; const aData: RawByteString; PhysicalIndexOldNew: PInt64=nil): integer; virtual;

*Update a record content in the database*
- in fact, a new invisible record is created, and an alias will map this new record to the original ID
- the physical replacement will take place only during Pack method call
- returns the new content ID value on success
- returns 0 on error

procedure Clear; virtual;

*Clear the whole table content and indexes*

procedure GetIterating(aCallBack: TSynBigTableIterateEvent; Order: TSynBigTableIterationOrder=ioPhysical; Opaque: pointer=nil; DontRetrieveData: Boolean=false);

*Call a supplied Event method by iterating through all table items*
- Event will be called following the physical order of data in the disk file (somewhat faster), or incremental ID depending of Order parameter
- Event can set the result to TRUE to break the iteration loop
- the Opaque parameter will be supplied to the callback Event
- set DontRetrieveData to TRUE if you don't need any data to be set in the callback, but only the ID (faster)

procedure Pack(forceFlushOnDisk: boolean=false);

*Pack the database, i.e. delete all formerly deleted ID from the disk*
- if forceFlushOnDisk is TRUE, data is forced to be saved on the disk (slower but allow ACID behavior of the database file)
procedure UpdateToFile(forceFlushOnDisk: boolean=false; dontReopenReadBuffer: boolean=false);

Save last added entries into the files
- do nothing is nothing is to be written (if forceAlwaysWrite is false)
- can be called from time to time, after checking CurrentInMemoryDataSize
- if forceFlushOnDisk is TRUE, data is forced to be saved on the disk (slower but allow ACID behavior of the database file)

property Count: integer read GetCount;

The entries count

property CurrentInMemoryDataSize: Int64 read fCurrentInMemoryDataSize;

Returns the current in memory data size (in bytes)
- i.e. the data size not written yet to the disk
- can be used to flush regularly the data to disk by calling UpdateToFile method when this value reach a certain limit

property FileName: TFileName read fFileName;

The associated filename storing the database

property FileSizeOnDisk: Int64 read GetFileSizeOnDisk;

Returns the current data size stored on disk

property NumericalID[Index: integer]: integer read GetID;

Read-only access to a numerical ID, from its index
- index is from NumericalID[0] to NumericalID[Count-1]
- follows the numerical ID order for TSynBigTable, and the alphabetical order of UTF-8 keys for TSynBigTableString
- return 0 in case of out of range index
- this method can be slow with TSynBigTable (if they are some deleted or updated items - just call the Pack method to improve speed); but with a TSynBigTableString instance, it will be always fast
- don't use it to loop through all items, but rather the dedicated GetIterating() or GetAllIDs() fast methods

property Offset[Index: integer]: Int64 read GetOffset;

Retrieve an offset for a specified physical ID
- read from either fOffset32[] either fOffset64[]

property OnAfterPack: TSynBigTableAfterPackEvent read fOnAfterPack write fOnAfterPack;

Event called after a pack, just before the UpdateToFile() call
- can be used to synchronized the field indexes, e.g.

TSynBigTableString = class(TSynBigTable)

A class to store huge amount of data, just specified by a string ID
- string ID are case-sensitive (important warning)
- string ID are of RawUTF8 type, so you must make explicit conversion in your program to the native generic string type - you can use our Utf8ToString() and StringToUtf8() functions, which work for all version of Delphi (from Delphi 6 up to XE)
function Add(const aData: RawByteString; const aID: RawUTF8; ForcedID: integer=0): integer; reintroduce;
   Add a data to the database, and its associated string ID
   - return the unique numerical ID created to identify this data
   - return 0 if the string ID is invalid (i.e. void or already used)

function Delete(aID: integer; PhysicalIndex: PInteger=nil): boolean; overload;
   Delete an entry from its numerical ID

function Delete(const aID: RawUTF8): boolean; reintroduce; overload;
   Delete an entry from its string ID

function Get(const aID: RawUTF8; var aData: RawByteString): boolean; overload;
   Retrieve a content, from a supplied string ID
   - return TRUE if found, FALSE if this ID was not existing (or deleted)
   - return the data into aData, or "" if ID was not existing (or deleted)

function GetAllIDs(var IDs: TIntegerDynArray; Order: TSynBigTableIterationOrder=ioPhysical): integer; override;
   Fast retrieval of all IDs
   - this overridden method handle ioFaster order, i.e: the fHeaderID[] content
   - returns the number of IDs stored in the integer array

function GetAsStream(const aID: RawUTF8): TStream; overload;
   Retrieve a content, from a supplied ID, into a TStream
   - if the ID is not correct, returns nil
   - if the ID is correct, returns a TStream instance, able to access to the associated content
   - in most cases, this TStream is just a wrapper around the memory mapped buffer in memory
   - the TStream must be consumed immediately, before any Pack or UpdateToFile method calls
   - the caller must Free the returned TStream instance
   - if the data is not already memory mapped (i.e. for files >= 2 GB) a custom
     TSynMemoryStreamMapped is used to access the data from disk

function GetPointer(const aID: RawUTF8; var aTempData: RawByteString; DataLen: PInteger=nil): pointer; overload;
   Retrieve a content, from a supplied ID, into a pointer
   - returns nil on error
   - returns a pointer to the data on success, directly from the memory mapped file on most cases;
     if the data is not in a memory mapped buffer (i.e. for files >= 2 GB) the aTempData variable is
     used to read the data from disk
   - in case of success, if DataLen is not nil, it will be filled with the corresponding data length
   - this method is therefore much faster than Get() for big size of data

function IDToString(aID: integer): RawUTF8;
   Retrieve the UTF-8 encoded string ID of a given numerical ID
   - return "" if this ID was not found

function StringToID(const aID: RawUTF8): integer;
   Retrieve a numerical ID from a UTF-8 encoded string ID
   - return 0 if this string ID was not found
function Update(const aData: RawByteString; const aID: RawUTF8): boolean;
    overload;

    Update a record content in the database
    - return true if the record was successfully updated

procedure Clear; override;

    Clear the whole table content and indexes

property StringID[Index: integer]: RawUTF8 read GetStringID;

    Read-only access to a string ID, from its index
    - index is from StringID[0] to StringID[Count-1]
    - string IDs are alphabetically sorted
    - return "" in case of out of range index

TSynBigTableTable = class(TSynBigTable)

    An abstract class, associating a TSynTable to a Big Table
    - use optimized TSynTable logic for handling field values, using our SBF compact binary format (similar to BSON or Protocol Buffers)

constructor Create(const aFileName: TFileName; const aTableName: RawUTF8;
    GetRecordData: TSynTableGetRecordData; FileOpenMode: Cardinal = 0); reintroduce;

    Initialize the database engine with a supplied filename
    - you can specify an internal Table Name, similar to SQL table name
    - you should better call either TSynBigTableMetaData or TSynBigTableRecord reintroduced constructor, which will set GetRecordData parameter as expected

destructor Destroy; override;

    Finalize memory, and save all content

function AddField(const aName: RawUTF8; aType: TSynTableFieldType; aOptions:
    TSynTableFieldOptions=[]): boolean;

    Add a field description to the table
    - just a wrapper to the Table.AddField method
    - warning: caller must call the AddFieldUpdate method when all AddField() methods have been called, in order to eventually process all already existing data to the resulting new field order
    - physical order does not necessary follow the AddField() call order: for better performance, it will try to store fixed-sized record first, multiple of 4 bytes first (access is faster if dat is 4 byte aligned), then variable-length after fixed-sized fields; in all case, a field indexed will be put first
    - if tfoUnique is set in aOptions and there is already some data, this method will raise an exception: it's not possible to have multiple void data unique, so it will always fail the constraint

function RecordGet(aID: integer): TSynTableData; overload;

    Retrieve a record as a TSynTableData to access its properties
    - using TSynTableData is faster than a TSynTableVariantType variant
**function** RecordUpdate(const aDataRecord: TSynTableData): boolean; **virtual**; **abstract**;

Update a Record from a given TSynTableData content
- using TSynTableData is faster than a TSynTableVariantType variant
- aRecord.ID is used to identify the record for calling raw Update()
- returns TRUE on success, FALSE on error (e.g. tftUnique constraint failure)
- for TSynBigTableMetaData, only update the metadata content, not the main record content

**function** Search(Field: TSynTableFieldProperties; const WhereValue: **variant**; var ResultID: TIntegerDynArray; var ResultIDCount: integer; Limit: Integer=0; ForceIterate: TSynBigTableIterationOrder=ioNone): boolean; overload;

Search for a matching value in a given field
- add the matching IDs in ResultID[] (in sorted order, with no duplicate)
- will use any existing index, or will iterate through all data (slower) if the ForceIterate parameter is either ioPhysical or ioID
- the Limit parameter is similar to the SQL LIMIT clause: if greater than 0, an upper bound on the number of rows returned is placed (e.g. set Limit=1 to only retrieve the first match)

**function** Search(Field: TSynTableFieldProperties; const WhereValue: **TSBFString**; var ResultID: TIntegerDynArray; var ResultIDCount: integer; Limit: Integer=0; ForceIterate: TSynBigTableIterationOrder=ioNone): boolean; overload;

Search for a matching value in a given field
- add the matching IDs in ResultID[] (in sorted order, with no duplicate), and update the number matching of elements in ResultIDCount (for performance reasons, the ResultID[] array remains filled with 0 until ResultID[ResultIDCount-1] itemk)
- will use any existing index, or will iterate through all data (slower) if the ForceIterate parameter is either ioPhysical or ioID
- the Limit parameter is similar to the SQL LIMIT clause: if greater than 0, an upper bound on the number of rows returned is placed (e.g. set Limit=1 to only retrieve the first match)

**function** VariantGet(aID: integer): **Variant**; **virtual**; **abstract**;

Retrieve a TSynTableVariantType variant to access a record properties

**function** VariantVoid: **Variant**;

Retrieve a void TSynTableVariantType variant instance
- similar to a call to Table.Data call

**procedure** AddFieldUpdate; **virtual**; **abstract**;

This method must be called after calls to AddField/Table.AddField methods
- this will launch the recreation of the database file content, if some field were effectively added (to map the new field layout): in this case some default void value is set for all newly added fields
- for TSynBigTableRecord, this method may recreate the field order then reload all field instances: you must retrieve all TSynTableFieldProperties instances after this method call via proper
  aField := Table.Table['FieldName'];

**procedure** Clear; **override**;

Clear the whole table content and indexes
- also delete the field layout
procedure RecordGet(aID: integer; var result: TSynTableData); overload; virtual;
abstract;

*Retrieve a record as a TSynTableData to access its properties*
- using TSynTableData is faster than a TSynTableVariantType variant
- this overloaded function doesn't use a function return, therefore will avoid a Record copy content (faster)

property Table: TSynTable read fTable;
*The associated field description*

property TableName: RawUTF8 read fTableName;
*The internal TableName*

TSynBigTableMetaData = class(TSynBigTableTable)

*A class to store huge data (like files content), with metadata fields associated with every record*
- this class will store the fields in memory, then uses TSynBigTable records to store some huge data blocks (e.g. file content), whereas TSynBigTableRecord will store the fields in the records:
TSynBigTableRecord is preferred for huge number of records, and TSynBigTableMetaData is designed for less number of records, but will natively handle associated "blob-like" data. For instance, TSynBigTableRecord would be the right class to implement a logging table, whereas TSynBigTableMetaData would be ideal for storing pictures.
- use optimized TSynTable logic for handling metadata field values, using our SBF compact binary format (similar to BSON or Protocol Buffers)
- you can access to any metadata fields by using a custom TSynTableVariantType variant type, allowing late-binding in the code (this method is slower than direct access to the data due to the Variant overhead, but is perhaps more convenient)

constructor Create(const aFileName: TFileName; const aTableName: RawUTF8; FileOpenMode: Cardinal = 0); reintroduce;
*Initialize the database engine with a supplied filename*
- you can specify an internal Table Name, similar to SQL table name

function Add(const aData: RawByteString; const aMetaData: TSBFString): integer; reintroduce; overload;
*Add a data item with its associated metadata record to the table*
- the metadata record uses our SBF enconding, and is mandatory
- returns the unique ID created to identify this data
- returns 0 on adding error (e.g. if a tftUnique constraint failed, or if the supplied aMetaData is void)

function Delete(aID: integer; PhysicalIndex: PInteger=nil): boolean; override;
*Overridden method to delete an entry from its numerical ID*
- this method will handle the metadata fields synchronization

function GetMetaDataFromID(aID: integer): pointer;
*Retrieve the metadata record of a given ID, encoded in our SBF format*
- it could be more convenient to use VariantGet() or even the faster RecordGet() methods
function RecordAdd(const aData: RawByteString; const aMetaDataRecord: TSynTableData): integer;

Add a record to the table, with associated meta data
- using TSynTableData is faster than a TSynTableVariantType variant
- return the unique ID created to identify this data
- returns 0 on adding error (e.g. if a tftUnique constraint failed)

function RecordUpdate(const aMetaDataRecord: TSynTableData): boolean; override;

Update a Record from a given TSynTableData content
- using TSynTableData is faster than a TSynTableVariantType variant
- aRecord.ID is used to identify the record for calling raw Update()
- returns TRUE on success, FALSE on error (e.g. tftUnique constraint failure)
- this method will only update the metadata - the main record data must be updated with the inherited Update() method

function Update(aID: integer; const aMetaData: TSBFString): boolean; reintroduce; overload;

Update a metadata record using our SBF encoding
- returns TRUE on success, FALSE on error (e.g. tftUnique constraint failure)
- this method will only update the metadata - the main record data must be updated with the inherited Update() method

function VariantAdd(const aData: RawByteString; const aMetaDataRecord: Variant): integer;

Add a data item with its associated metadata record to the table
- the metadata record is a TSynTableVariantType variant
- returns the unique ID created to identify this data
- returns 0 on adding error (e.g. if a tftUnique constraint failed)

function VariantGet(aID: integer): Variant; override;

Retrieve a TSynTableVariantType variant to access a record metadata

function VariantUpdate(const aMetaDataRecord: Variant): boolean;

Update a metadata record as TSynTableVariantType variant
- aRecord.ID is used to identify the record for calling raw Update()
- returns TRUE on success, FALSE on error (e.g. tftUnique constraint failure, or wrong variant type)
- this method will only update the metadata - the main record data must be updated with the inherited Update() method

procedure AddFieldUpdate; override;

This method must be called after calls to AddField/Table.AddField methods
- this will launch the recreation of the database file content, if some field were effectively added
  (to map the new field layout): in this case some default void value is set for all newly added fields
- for TSynBigTableMeta, this method may recreate the field order, but won't change the TSynTableFieldProperties instances
procedure RecordGet(aID: integer; var result: TSynTableData); overload; override;

Retrieve a record as a TSynTableData to access its properties
- using TSynTableData is faster than a TSynTableVariantType variant
- this overloaded function doesn’t use a function return, therefore will avoid a Record copy content (faster)

TSynBigTableRecord = class(TSynBigTableTable)

A class to store huge amount of data, with fields in every record
- this class will store the fields in the TSynBigTable records, whereas TSynBigTableMetaData will store the fields in memory, and will uses records to store some huge data blocks (e.g. file content): TSynBigTableRecord is preferred for huge number of records, and TSynBigTableMetaData is designed for less records, but with associated "blob-like" data. For instance, TSynBigTableRecord would be the right class to implement a logging table, whereas TSynBigTableMetaData would be ideal for storing pictures.
- use optimized TSynTable logic for handling field values, using our SBF compact binary format (similar to BSON or Protocol Buffers)
- you can access to any record content fields by using a custom TSynTableVariantType variant type, allowing late-binding in the code (this method is slower than direct access to the data due to the Variant overhead, but is perhaps more convenient)

constructor Create(const aFileName: TFileName; const aTableName: RawUTF8; FileOpenMode: Cardinal = 0); reintroduce;

Initialize the database engine with a supplied filename
- you can specify an internal Table Name, similar to SQL table name

function Add(const aData: RawByteString; ForcedID: integer=0; PhysicalIndex: PInteger=nil; OldPhysicalIndex: integer=-1): integer;

Overridden method to add a record to the database
- this method will handle the field indexes synchronization
- returns 0 on adding error (e.g. if a tftUnique constraint failed)

function Delete(aID: integer; PhysicalIndex: PInteger=nil): boolean;

Overridden method to delete an entry from its numerical ID
- this method will handle the field indexes synchronization

function RecordAdd(const aRecord: TSynTableData; aForcedID: integer=0): integer;

Add a record to the table
- using TSynTableData is faster than a TSynTableVariantType variant
- return the unique ID created to identify this data
- you can specify an expected ID to be used in aForceID parameter
- returns 0 on adding error (e.g. if a tftUnique constraint failed)

function RecordUpdate(const aRecord: TSynTableData): boolean; override;

Update a Record from a given TSynTableData content
- using TSynTableData is faster than a TSynTableVariantType variant
- aRecord.ID is used to identify the record for calling raw Update()
- returns TRUE on success, FALSE on error (e.g. tftUnique constraint failure)
function Update(aID: Integer; const aData: RawByteString; PhysicalIndexOldNew: PInt64=nil): integer; override;

Overridden method to update a record content in the database
- returns 0 on updating error (e.g. if a tftUnique constraint failed)

function VariantAdd(const aRecord: Variant): integer;
Add a record to the table
- the record is a TSynTableVariantType variant
- returns the unique ID created to identify this data
- returns 0 on adding error (e.g. if a tftUnique constraint failed)

function VariantGet(aID: integer): Variant; override;
Retrieve a TSynTableVariantType variant to access a record properties

function VariantUpdate(const aRecord: Variant): boolean;
Update a TSynTableVariantType variant
- aRecord.ID is used to identify the record for calling raw Update()
- returns TRUE on success, FALSE on error (e.g. tftUnique constraint failure)

procedure AddFieldUpdate; override;

This method must be called after calls to AddField/Table.AddField methods
- this will launch the recreation of the database file content, if some field were effectively added (to map the new field layout): in this case some default void value is set for all newly added fields
- for TSynBigTableRecord, this method may recreate the field order then reload all field instances: you must retrieve all TSynTableFieldProperties instances after this method call via proper
  aField := Table.Table['FieldName'];

procedure RecordGet(aID: integer; var result: TSynTableData); overload; override;
Retrieve a record as a TSynTableData to access its properties
- using TSynTableData is faster than a TSynTableVariantType variant
- this overloaded function doesn't use a function return, therefore will avoid a Record copy content (faster)

TTestBigTable = class(TSynTestCase)
  Unitary testing of the SynBigTable unit

procedure _TSynBigTable;
  Test TSynBigTable class

procedure _TSynBigTableMetaData;
  Test TSynBigTableMetaData class

procedure _TSynBigTableRecord;
  Test TSynBigTableRecord class

procedure _TSynBigTableString;
  Test TSynBigTableString class

Types implemented in the SynBigTable unit
**TSynBigTableAfterPackEvent** = `procedure(var NewIndexes: TIntegerDynArray) of object;`

*Event called after a pack, just before the UpdateToFile()*
- can be used to synchronize the field indexes, e.g.
- format of supplied parameter is NewIndexes[oldIndex] := newIndex

**TSynBigTableCustomHeader** = ( sbtRead, sbtBeforeWrite, sbtWrite, sbtAfterRead );

*Possible actions for CustomHeader() protected virtual method*

**TSynBigTableIterateEvent** = `function(Sender: TObject; Opaque: pointer; ID, DataIndex: integer; Data: pointer; DataLen: integer): boolean of object;`

*Prototype of a callback function for iterating through all items of a table*
- will be called following the incremental ID order
- implementation can set the result to TRUE to break the iteration loop
- the Data pointer is either direct access to the direct mapped buffer, or a global temporary buffer: use the data between two iterations, but copy the content if you need some persistency
- the DataIndex parameter is the index of the data, physically on disk

**TSynBigTableIterationOrder** = ( ioNone, ioPhysical, ioID, ioFaster, ioInternalID, ioInternalPhysical );

*The way the GetIterating() method will loop through all items*
- ioInternalID and ioInternalPhysical are internal codes used by GetID and GetAllPhysicalIndexes methods

**Constants implemented in the SynBigTable unit**

`BIGTABLE_AUTOFLUSH_SIZE = 1 shl 28;`

*Will flush the in-memory data to disk when reached 256 MB of data in RAM*

**Functions or procedures implemented in the SynBigTable unit**

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestBigTable</td>
<td>Unitary test function of the TSynBigTable class</td>
<td>716</td>
</tr>
</tbody>
</table>

**function TestBigTable: boolean;**

*Unitary test function of the TSynBigTable class*
- return TRUE if test was OK, FALSE on any error
27.5. SynCommons.pas unit

Purpose: Common functions used by most Synopse projects
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The SynCommons unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.1.2</td>
<td>UTF-8 JSON format shall be used to communicate</td>
<td>2545</td>
</tr>
<tr>
<td>DI-2.1.4</td>
<td>The framework shall provide some Cross-Cutting components</td>
<td>2547</td>
</tr>
</tbody>
</table>

Units used in the SynCommons unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynLZ</td>
<td>SynLZ Compression routines</td>
<td>1393</td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>
SynCommons class hierarchy

Objects implemented in the SynCommons unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDocVariant</td>
<td>Exception class associated to TDocVariant JSON/BSON document</td>
<td>812</td>
</tr>
<tr>
<td>EFastReader</td>
<td>Exception raised during TFastReader decoding</td>
<td>812</td>
</tr>
<tr>
<td>ESynException</td>
<td>Generic parent class of all custom Exception types of this unit</td>
<td>811</td>
</tr>
<tr>
<td>IAutoFree</td>
<td>Interface for TAutoFree to register another TObject instance to an existing IAutoFree local variable</td>
<td>855</td>
</tr>
<tr>
<td>IAutoLocker</td>
<td>An interface used by TAutoLocker to protect multi-thread execution</td>
<td>857</td>
</tr>
<tr>
<td>ILocalPrecisionTimer</td>
<td>Interface to a reference counted high resolution timer instance</td>
<td>848</td>
</tr>
<tr>
<td>ILockedDocVariant</td>
<td>Internal error C3517 under Delphi 5 : ref-counted interface for thread-safe access to a TDocVariant document</td>
<td>860</td>
</tr>
<tr>
<td>IObjectDynArray</td>
<td>Defines a wrapper interface around a dynamic array of TObject</td>
<td>750</td>
</tr>
<tr>
<td>TAlgoCompress</td>
<td>Abstract low-level parent class for generic compression/decompression algorithms</td>
<td>796</td>
</tr>
<tr>
<td>TAlgoCompressWithNoDestLen</td>
<td>Abstract class storing the plain length before calling compression API</td>
<td>800</td>
</tr>
<tr>
<td>TAlgoSynLZ</td>
<td>Implement our fast SynLZ compression as a TAlgoCompress class</td>
<td>800</td>
</tr>
<tr>
<td>TAutoFree</td>
<td>Simple reference-counted storage for local objects</td>
<td>856</td>
</tr>
<tr>
<td>TAutoLocker</td>
<td>Reference-counted block code critical section</td>
<td>858</td>
</tr>
<tr>
<td>TDiv100Rec</td>
<td>Small structure used as convenient result to Div100() procedure</td>
<td>729</td>
</tr>
<tr>
<td>TDocVariant</td>
<td>A custom variant type used to store any JSON/BSON document-based content</td>
<td>823</td>
</tr>
<tr>
<td>TDocVariantData</td>
<td>Memory structure used for TDocVariant storage of any JSON/BSON document-based content as variant</td>
<td>828</td>
</tr>
<tr>
<td>TDWordRec</td>
<td>Binary access to an unsigned 32-bit value (4 bytes in memory)</td>
<td>812</td>
</tr>
<tr>
<td>TDynArray</td>
<td>A wrapper around a dynamic array with one dimension</td>
<td>731</td>
</tr>
<tr>
<td>TDynArrayHashed</td>
<td>Used to access any dynamic array elements using fast hash</td>
<td>746</td>
</tr>
<tr>
<td>TDynArrayHasher</td>
<td>Allow O(1) lookup to any dynamic array content</td>
<td>745</td>
</tr>
<tr>
<td>TDynArrayLoadFrom</td>
<td>Allows to iterate over a TDynArray.SaveTo binary buffer</td>
<td>744</td>
</tr>
<tr>
<td>TExeVersion</td>
<td>Stores some global information about the current executable and computer</td>
<td>820</td>
</tr>
<tr>
<td>TFakeWriterStream</td>
<td>A fake TStream, which will just count the number of bytes written</td>
<td>808</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TFileVersion</td>
<td>To have existing RTTI for published properties used to retrieve version information from any EXE</td>
<td>817</td>
</tr>
<tr>
<td>TFindFiles</td>
<td>File found result item, as returned by FindFiles()</td>
<td>729</td>
</tr>
<tr>
<td>THash128Rec</td>
<td>Map a 128-bit hash as an array of lower bit size values</td>
<td>812</td>
</tr>
<tr>
<td>THash256Rec</td>
<td>Map a 256-bit hash as an array of lower bit size values</td>
<td>812</td>
</tr>
<tr>
<td>THash512Rec</td>
<td>Map a 512-bit hash as an array of lower bit size values</td>
<td>812</td>
</tr>
<tr>
<td>THeapMemoryStream</td>
<td>To be used instead of TMemoryStream, for speed</td>
<td>821</td>
</tr>
<tr>
<td>TInterfacedObjectLocked</td>
<td>Adding locking methods to a TInterfacedObject with virtual constructor</td>
<td>758</td>
</tr>
<tr>
<td>TInterfacedObjectWithCustomCreate</td>
<td>Abstract parent class with threadsafe implementation of IInterface and a virtual constructor</td>
<td>752</td>
</tr>
<tr>
<td>TJSONCustomParserCustom</td>
<td>Used to store additional RTTI as a ptCustom kind of property</td>
<td>766</td>
</tr>
<tr>
<td>TJSONCustomParserCustomRecord</td>
<td>Implement a reference to a registered record type</td>
<td>767</td>
</tr>
<tr>
<td>TJSONCustomParserCustomSimple</td>
<td>Used to store additional RTTI for simple type as a ptCustom kind</td>
<td>766</td>
</tr>
<tr>
<td>TJSONCustomParserRTTI</td>
<td>Used to store additional RTTI in TJSONCustomParser internal structures</td>
<td>764</td>
</tr>
<tr>
<td>TJSONRecordAbstract</td>
<td>Used to handle additional RTTI for JSON record serialization</td>
<td>767</td>
</tr>
<tr>
<td>TJSONRecordRTTI</td>
<td>Used to handle JSON record serialization using RTTI</td>
<td>768</td>
</tr>
<tr>
<td>TJSONRecordTextDefinition</td>
<td>Used to handle text-defined additional RTTI for JSON record serialization</td>
<td>768</td>
</tr>
<tr>
<td>TLecuyer</td>
<td>Low-level object implementing a 32-bit Pierre L’Ecuyer software generator</td>
<td>764</td>
</tr>
<tr>
<td>TLocalPrecisionTimer</td>
<td>Reference counted high resolution timer (for accurate speed statistics)</td>
<td>849</td>
</tr>
<tr>
<td>TLockedDocVariant</td>
<td>Allows thread-safe access to a TDocVariant document</td>
<td>861</td>
</tr>
<tr>
<td>TMemoryMap</td>
<td>Handle memory mapping of a file content</td>
<td>806</td>
</tr>
<tr>
<td>TMemoryMapText</td>
<td>Able to read a UTF-8 text file using memory map</td>
<td>806</td>
</tr>
<tr>
<td>TMultiPart</td>
<td>Used by MultiPartFormDataDecode() to return one item of its data</td>
<td>730</td>
</tr>
<tr>
<td>TNameValueUTF8Char</td>
<td>Store one name/value pair of raw UTF-8 content, from a JSON buffer</td>
<td>810</td>
</tr>
<tr>
<td>TObjectDynArrayWrapper</td>
<td>A wrapper to own a dynamic array of TObject</td>
<td>751</td>
</tr>
<tr>
<td>TObjectListHashed</td>
<td>This class behaves like TList/TObjectList, but will use hashing for (much) faster IndexOf() method</td>
<td>788</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TObjectListHashedAbstract</td>
<td>Abstract ancestor to manage a dynamic array of TObject</td>
<td>788</td>
</tr>
<tr>
<td>TObjectListPropertyHashed</td>
<td>This class will hash and search for a sub property of the stored objects</td>
<td>789</td>
</tr>
<tr>
<td>TOperatingSystemVersion</td>
<td>The running Operating System, encoded as a 32-bit integer</td>
<td>819</td>
</tr>
<tr>
<td>TOSVersionInfoEx</td>
<td>Low-level API structure, not defined in older Delphi versions</td>
<td>819</td>
</tr>
<tr>
<td>TPersistentWithCustomCreate</td>
<td>Abstract parent class with a virtual constructor, ready to be overridden</td>
<td>752</td>
</tr>
<tr>
<td>TPointerClassHash</td>
<td>Handle a O(1) hashed-based storage of TPointerClassHashed, from a pointer</td>
<td>790</td>
</tr>
<tr>
<td>TPointerClassHashed</td>
<td>Abstract class stored by a TPointerClassHash list</td>
<td>789</td>
</tr>
<tr>
<td>TPointerClassHashLock</td>
<td>Handle a O(1) hashed-based storage of TPointerClassHashed, from a pointer</td>
<td>790</td>
</tr>
<tr>
<td>TPrecisionTimer</td>
<td>High resolution timer (for accurate speed statistics)</td>
<td>846</td>
</tr>
<tr>
<td>TPublishedMethodInfo</td>
<td>Information about one method, as returned by GetPublishedMethods</td>
<td>730</td>
</tr>
<tr>
<td>TQWordRec</td>
<td>Binary access to an unsigned 64-bit value (8 bytes in memory)</td>
<td>812</td>
</tr>
<tr>
<td>TRawByteStringStream</td>
<td>A TStream using a RawByteString as internal storage</td>
<td>808</td>
</tr>
<tr>
<td>TRawUTF8Interning</td>
<td>Allow to store only one copy of distinct RawUTF8 values</td>
<td>759</td>
</tr>
<tr>
<td>TRawUTF8InterningSlot</td>
<td>Used to store one list of hashed RawUTF8 in TRawUTF8Interning pool</td>
<td>759</td>
</tr>
<tr>
<td>TRawUTF8List</td>
<td>TStringList-class optimized to work with our native UTF-8 string type</td>
<td>792</td>
</tr>
<tr>
<td>TSortedIntegerArray</td>
<td>Used to store and retrieve Integers in a sorted array</td>
<td>730</td>
</tr>
<tr>
<td>TSortedWordArray</td>
<td>Used to store and retrieve Words in a sorted array</td>
<td>730</td>
</tr>
<tr>
<td>TSynAnsiConvert</td>
<td>An abstract class to handle Ansi to/from Unicode translation</td>
<td>723</td>
</tr>
<tr>
<td>TSynAnsiFixedWidth</td>
<td>A class to handle Ansi to/from Unicode translation of fixed width encoding (i.e. non MBCS)</td>
<td>725</td>
</tr>
<tr>
<td>TSynAnsiUTF16</td>
<td>A class to handle UTF-16 to/from Unicode translation</td>
<td>727</td>
</tr>
<tr>
<td>TSynAnsiUTF8</td>
<td>A class to handle UTF-8 to/from Unicode translation</td>
<td>726</td>
</tr>
<tr>
<td>TSynDate</td>
<td>A simple way to store a date as Year/Month/Day</td>
<td>812</td>
</tr>
<tr>
<td>TSynDictionary</td>
<td>Thread-safe dictionary to store some values from associated keys</td>
<td>801</td>
</tr>
<tr>
<td>TSynFPUException</td>
<td>A simple class which will set FPU exception flags for a code block</td>
<td>855</td>
</tr>
<tr>
<td>TSynInterfacedObject</td>
<td>An abstract ancestor, for implementing a custom TInterfacedObject like class</td>
<td>855</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TSynInvokeableVariantType</td>
<td>Custom variant handler with easier/faster access of variant properties, and JSON serialization support</td>
<td>821</td>
</tr>
<tr>
<td>TSynList</td>
<td>Simple and efficient TList, without any notification</td>
<td>753</td>
</tr>
<tr>
<td>TSynLocker</td>
<td>Allow to add cross-platform locking methods to any class instance</td>
<td>755</td>
</tr>
<tr>
<td>TSynLogExceptionContext</td>
<td>Calling context of TSynLogExceptionToStr callbacks</td>
<td>810</td>
</tr>
<tr>
<td>TSynMemoryStream</td>
<td>A TStream pointing to some in-memory data, for instance UTF-8 text</td>
<td>809</td>
</tr>
<tr>
<td>TSynMemoryStreamMapped</td>
<td>A TStream created from a file content, using fast memory mapping</td>
<td>809</td>
</tr>
<tr>
<td>TSynMonitor</td>
<td>A generic value object able to handle any task / process statistic</td>
<td>850</td>
</tr>
<tr>
<td>TSynMonitorInputOutput</td>
<td>Handle generic process statistic with an incoming and outgoing processing data size and bandwidth</td>
<td>853</td>
</tr>
<tr>
<td>TSynMonitorOneSize</td>
<td>Able to serialize any immediate size as bytes number</td>
<td>850</td>
</tr>
<tr>
<td>TSynMonitorOneTime</td>
<td>Able to serialize any immediate timing as raw micro-seconds number or text</td>
<td>849</td>
</tr>
<tr>
<td>TSynMonitorServer</td>
<td>Could monitor a standard Server</td>
<td>854</td>
</tr>
<tr>
<td>TSynMonitorSize</td>
<td>Able to serialize any cumulative size as bytes number</td>
<td>850</td>
</tr>
<tr>
<td>TSynMonitorThroughput</td>
<td>Able to serialize any bandwidth as bytes count per second</td>
<td>850</td>
</tr>
<tr>
<td>TSynMonitorTime</td>
<td>Able to serialize any cumulative timing as raw micro-seconds number or text</td>
<td>849</td>
</tr>
<tr>
<td>TSynMonitorWithSize</td>
<td>Handle generic process statistic with a processing data size and bandwidth</td>
<td>853</td>
</tr>
<tr>
<td>TSynNameValue</td>
<td>Pseudo-class used to store Name/Value RawUTF8 pairs</td>
<td>761</td>
</tr>
<tr>
<td>TSynNameValueItem</td>
<td>Store one Name/Value pair, as used by TSynNameValue class</td>
<td>761</td>
</tr>
<tr>
<td>TSynObjectList</td>
<td>Simple and efficient TObjectList, without any notification</td>
<td>754</td>
</tr>
<tr>
<td>TSynObjectListLocked</td>
<td>Add locking methods to a TSynObjectList</td>
<td>791</td>
</tr>
<tr>
<td>TSynPersistent</td>
<td>Our own empowered TPersistent-like parent class</td>
<td>753</td>
</tr>
<tr>
<td>TSynPersistentLock</td>
<td>Adding locking methods to a TSynPersistent with virtual constructor</td>
<td>758</td>
</tr>
<tr>
<td>TSynPersistentLocked</td>
<td>Used for backward compatibility only with existing code</td>
<td>758</td>
</tr>
<tr>
<td>TSynSystemTime</td>
<td>A cross-platform and cross-compiler TSystemTime 128-bit structure</td>
<td>813</td>
</tr>
<tr>
<td>TTempUTF8</td>
<td>Implements a stack-based storage of some (UTF-8 or binary) text</td>
<td>728</td>
</tr>
<tr>
<td>TTextWriter</td>
<td>A memory structure which avoids a temporary RawUTF8 allocation</td>
<td>729</td>
</tr>
<tr>
<td></td>
<td>Simple writer to a Stream, specialized for the TEXT format</td>
<td>769</td>
</tr>
</tbody>
</table>
# Synopse mORMot Framework

**Software Architecture Design 1.18**  
**Date:** September 16, 2020

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTextWriterWithEcho</td>
<td>Stream TEXT writer, with optional echoing of the lines</td>
<td>787</td>
</tr>
<tr>
<td>TTimeLogBits</td>
<td>Internal memory structure for direct access to a TTimeLog type value</td>
<td>815</td>
</tr>
<tr>
<td>TValueUTF8Char</td>
<td>Points to one value of raw UTF-8 content, decoded from a JSON buffer</td>
<td>809</td>
</tr>
<tr>
<td>TValueResult</td>
<td>Kind of result returned by FromVarBlob() function</td>
<td>731</td>
</tr>
<tr>
<td>TWinRegistry</td>
<td>Direct access to the Windows Registry</td>
<td>819</td>
</tr>
</tbody>
</table>

**TSynAnsiConvert = class(TObject)**

_An abstract class to handle Ansi to/from Unicode translation_

- implementations of this class will handle efficiently all Code Pages
- this default implementation will use the Operating System APIs
- you should not create your own class instance by yourself, but should better retrieve an instance using TSynAnsiConvert.Engine(), which will initialize either a TSynAnsiFixedWidth or a TSynAnsiConvert instance on need

**constructor** Create(aCodePage: cardinal); reintroduce; virtual;  
Initialize the internal conversion engine

**function** AnsiBufferToRawUTF8(Source: PAnsiChar; SourceChars: Cardinal): RawUTF8; overload; virtual;  
Direct conversion of a PAnsiChar buffer into a UTF-8 encoded string  
- Dest^ buffer must be reserved with at least SourceChars*2 bytes  
- this default implementation will use the Operating System APIs  
- will append a trailing #0 to the returned PWideChar, unless NoTrailingZero is set

**function** AnsiBufferToUnicode(Dest: PWideChar; Source: PAnsiChar; SourceChars: Cardinal; NoTrailingZero: boolean=false): PWideChar; overload; virtual;  
Direct conversion of a PAnsiChar buffer into an Unicode buffer  
- Dest^ buffer must be reserved with at least SourceChars*2 bytes  
- this default implementation will use the Operating System APIs  
- will append a trailing #0 to the returned PWideChar, unless NoTrailingZero is set

**function** AnsiBufferToUTF8(Dest: PUTF8Char; Source: PAnsiChar; SourceChars: Cardinal; NoTrailingZero: boolean=false): PUTF8Char; overload; virtual;  
Direct conversion of a PAnsiChar buffer into a UTF-8 encoded buffer  
- Dest^ buffer must be reserved with at least SourceChars*2 bytes  
- this default implementation will use the Operating System APIs  
- will append a trailing #0 to the returned PUTF8Char, unless NoTrailingZero is set

**function** AnsiToAnsi(From: TSynAnsiConvert; const Source: RawByteString): RawByteString; overload;  
Convert any Ansi Text (providing a From converted) into Ansi Text

**function** AnsiToAnsi(From: TSynAnsiConvert; Source: PAnsiChar; SourceChars: Cardinal): RawByteString; overload;  
Convert any Ansi buffer (providing a From converted) into Ansi Text

**function** AnsiToRawUnicode(const AnsiText: RawByteString): RawUnicode; overload;  
Convert any Ansi Text into a UTF-16 Unicode String  
- returns a value using our RawUnicode kind of string
### Function `AnsiToRawUnicode` (Source: PAnsiChar; SourceChars: Cardinal): RawUnicode; overload; **virtual**
- Convert any Ansi buffer into an Unicode String
- returns a value using our RawUnicode kind of string

### Function `AnsiToUnicodeString` (const Source: RawByteString): SynUnicode; overload;
- Convert any Ansi buffer into an Unicode String
- returns a SynUnicode, i.e. Delphi 2009+ UnicodeString or a WideString

### Function `AnsiToUnicodeString` (Source: PAnsiChar; SourceChars: Cardinal): SynUnicode; overload;
- Convert any Ansi buffer into an Unicode String
- returns a SynUnicode, i.e. Delphi 2009+ UnicodeString or a WideString

### Function `AnsiToUTF8` (const AnsiText: RawByteString): RawUTF8; **virtual**
- Convert any Ansi Text into an UTF-8 encoded String
- internaly calls AnsiBufferToUTF8 virtual method

### Class Function `Engine` (aCodePage: cardinal): TSynAnsiConvert;
- Returns the engine corresponding to a given code page
- a global list of TSynAnsiConvert instances is handled by the unit - therefore, caller should not release the returned instance
- will return nil in case of unhandled code page
- is aCodePage is 0, will return CurrentAnsiConvert value

### Function `RawUnicodeToAnsi` (const Source: RawUnicode): RawByteString;
- Convert any Unicode-encoded String into Ansi Text
- internaly calls UnicodeBufferToAnsi virtual method

### Function `UnicodeBufferToAnsi` (Dest: PAnsiChar; Source: PWideChar; SourceChars: Cardinal): PAnsiChar; overload; **virtual**;
- Direct conversion of an Unicode buffer into a PAnsiChar buffer
- Dest^ buffer must be reserved with at least SourceChars*3 bytes
- this default implementation will rely on the Operating System for all non ASCII-7 chars

### Function `UTF8BufferToAnsi` (Dest: PAnsiChar; Source: PUTF8Char; SourceChars: Cardinal): RawByteString; overload; **virtual**;
- Direct conversion of an UTF-8 encoded buffer into a PAnsiChar buffer
- no trailing #0 is appended to the buffer

### Function `UTF8ToAnsi` (const UTF8: RawUTF8): RawByteString; **virtual**
- Convert any UTF-8 encoded String into Ansi Text
- internaly calls UTF8BufferToAnsi virtual method
function Utf8ToAnsiBuffer(const S: RawUTF8; Dest: PAnsiChar; DestSize: integer): integer;

Direct conversion of a UTF-8 encoded string into a WinAnsi buffer
- will truncate the destination string to DestSize bytes (including the trailing #0), with a maximum handled size of 2048 bytes
- returns the number of bytes stored in Dest^ (i.e. the position of #0)

procedure UTF8BufferToAnsi(Source: PUTF8Char; SourceChars: Cardinal; var result: RawByteString); overload; virtual;

Convert any UTF-8 encoded buffer into Ansi Text
- internally calls UTF8BufferToAnsi virtual method

property CodePage: Cardinal read fCodePage;

Corresponding code page

TSynAnsiFixedWidth = class(TSynAnsiConvert)

A class to handle Ansi to/from Unicode translation of fixed width encoding (i.e. non MBCS)
- this class will handle efficiently all Code Page availables without MBCS encoding - like WinAnsi (1252) or Russian (1251)
- it will use internal fast look-up tables for such encodings
- this class could take some time to generate, and will consume more than 64 KB of memory: you should not create your own class instance by yourself, but should better retrieve an instance using TSynAnsiConvert.Engine(), which will initialize either a TSynAnsiFixedWidth or a TSynAnsiConvert instance on need
- this class has some additional methods (e.g. IsValid*) which take advantage of the internal lookup tables to provide some fast process

constructor Create(aCodePage: cardinal); override;

Initialize the internal conversion engine

function AnsiBufferToUnicode(Dest: PWideChar; Source: PAnsiChar; SourceChars: Cardinal; NoTrailingZero: boolean=false): PWideChar; override;

Direct conversion of a PAnsiChar buffer into an Unicode buffer
- Dest^ buffer must be reserved with at least SourceChars*2 bytes
- will append a trailing #0 to the returned PWideChar, unless NoTrailingZero is set

function AnsiBufferToUTF8(Dest: PUTF8Char; Source: PAnsiChar; SourceChars: Cardinal; NoTrailingZero: boolean=false): PUTF8Char; override;

Direct conversion of a PAnsiChar buffer into a UTF-8 encoded buffer
- Dest^ buffer must be reserved with at least SourceChars*3 bytes
- will append a trailing #0 to the returned PUTF8Char, unless NoTrailingZero is set

function AnsiToRawUnicode(Source: PAnsiChar; SourceChars: Cardinal): RawUnicode; override;

Convert any Ansi buffer into an Unicode String
- returns a value using our RawUnicode kind of string

function IsValidAnsi(WideText: PWideChar; Length: PtrInt): boolean; overload;

Return TRUE if the supplied unicode buffer only contains characters of the corresponding Ansi code page
- i.e. if the text can be displayed using this code page
function IsValidAnsi(WideText: PWideChar): boolean; overload;
Return TRUE if the supplied unicode buffer only contains characters of the corresponding Ansi code page
- i.e. if the text can be displayed using this code page

function IsValidAnsiUTF8Text: PUTF8Char): boolean;
Return TRUE if the supplied UTF-8 buffer only contains characters of the corresponding Ansi code page
- i.e. if the text can be displayed using this code page

function IsValidAnsiU8Bit.UTF8Text: PUTF8Char): boolean;
Return TRUE if the supplied UTF-8 buffer only contains 8 bits characters of the corresponding Ansi code page
- i.e. if the text can be displayed with only 8 bit unicode characters (e.g. no "tm" or such) within this code page

function UnicodeBufferToAnsi(Dest: PAnsiChar; Source: PWideChar; SourceChars: Cardinal): PAnsiChar; override;
Direct conversion of an Unicode buffer into a PAnsiChar buffer
- Dest^ buffer must be reserved with at least SourceChars*3 bytes
- this overridden version will use internal lookup tables for fast process

function UTF8BufferToAnsi(Dest: PAnsiChar; Source: PUTF8Char; SourceChars: Cardinal): PAnsiChar; override;
Direct conversion of an UTF-8 encoded buffer into a PAnsiChar buffer
- Dest^ buffer must be reserved with at least SourceChars bytes
- no trailing #0 is appended to the buffer

function WideCharToAnsiChar(wc: cardinal): integer;
Conversion of a wide char into the corresponding Ansi character
- return -1 for an unknown WideChar in the current code page

property AnsiToWorld: TWordDynArray read fAnsiToWorld;
Direct access to the Ansi-To-Unicode lookup table
- use this array like AnsiToWorld: array[byte] of word

property WideToAnsi: TByteDynArray read fWideToAnsi;
Direct access to the Unicode-To-Ansi lookup table
- use this array like WideToAnsi: array[word] of byte
- any unhandled WideChar will return ord(?)

TSynAnsiUTF8 = class(TSynAnsiConvert)
A class to handle UTF-8 to/from Unicode translation
- match the TSynAnsiConvert signature, for code page CP_UTF8
- this class is mostly a non-operation for conversion to/from UTF-8

constructor Create(aCodePage: cardinal); override;
Initialize the internal conversion engine

function AnsiBufferToRawUTF8(Source: PAnsiChar; SourceChars: Cardinal): RawUTF8; override;
Direct conversion of a PAnsiChar buffer into a UTF-8 encoded string
function AnsiBufferToUnicode(Dest: PWideChar; Source: PAnsiChar; SourceChars: Cardinal; NoTrailingZero: boolean=false): PWideChar; override;

*Direct conversion of a PAnsiChar UTF-8 buffer into an Unicode buffer*
- Dest^ buffer must be reserved with at least SourceChars*2 bytes
- will append a trailing #0 to the returned PWideChar, unless NoTrailingZero is set

function AnsiBufferToUTF8(Dest: PUTF8Char; Source: PAnsiChar; SourceChars: Cardinal; NoTrailingZero: boolean=false): PUTF8Char; override;

*Direct conversion of a PAnsiChar UTF-8 buffer into a UTF-8 encoded buffer*
- Dest^ buffer must be reserved with at least SourceChars*3 bytes
- will append a trailing #0 to the returned PUTF8Char, unless NoTrailingZero is set

function AnsiToRawUnicode(Source: PAnsiChar; SourceChars: Cardinal): RawUnicode; override;

*Convert any UTF-8 Ansi buffer into an Unicode String*
- returns a value using our RawUnicode kind of string

function AnsiToUTF8(const AnsiText: RawByteString): RawUTF8; override;

*Convert any Ansi Text into an UTF-8 encoded String*
- directly assign the input as result, since no conversion is needed

function UnicodeBufferToAnsi(Source: PWideChar; SourceChars: Cardinal): RawByteString; override;

*Direct conversion of an Unicode buffer into an Ansi Text*

function UnicodeBufferToAnsi(Dest: PAnsiChar; Source: PWideChar; SourceChars: Cardinal): PAnsiChar; override;

*Direct conversion of an Unicode buffer into a PAnsiChar UTF-8 buffer*
- Dest^ buffer must be reserved with at least SourceChars*3 bytes

function UTF8BufferToAnsi(Dest: PAnsiChar; Source: PUTF8Char; SourceChars: Cardinal): PAnsiChar; override;

*Direct conversion of an UTF-8 encoded buffer into a PAnsiChar UTF-8 buffer*
- Dest^ buffer must be reserved with at least SourceChars bytes
- no trailing #0 is appended to the buffer

function UTF8ToAnsi(const UTF8: RawUTF8): RawByteString; override;

*Convert any UTF-8 encoded String into Ansi Text*
- directly assign the input as result, since no conversion is needed

procedure UTF8BufferToAnsi(Source: PUTF8Char; SourceChars: Cardinal; var result: RawByteString); override;

*Convert any UTF-8 encoded buffer into Ansi Text*

TSynAnsiUTF16 = class(TSynAnsiConvert)

*A class to handle UTF-16 to/from Unicode translation*
- match the TSynAnsiConvert signature, for code page CP_UTF16
- even if UTF-16 is not an Ansi format, code page CP_UTF16 may have been used to store UTF-16 encoded binary content
- this class is mostly a non-operation for conversion to/from Unicode

constructor Create(aCodePage: cardinal); override;

*Initialize the internal conversion engine*
function AnsiBufferToUnicode(Dest: PWideChar; Source: PAnsiChar; SourceChars: Cardinal; NoTrailingZero: boolean=false): PWideChar; override;

Direct conversion of a PAnsiChar UTF-16 buffer into an Unicode buffer
- Dest^ buffer must be reserved with at least SourceChars*2 bytes
- will append a trailing #0 to the returned PWideChar, unless NoTrailingZero is set

function AnsiBufferToUTF8(Dest: PUTF8Char; Source: PAnsiChar; SourceChars: Cardinal; NoTrailingZero: boolean=false): PUTF8Char; override;

Direct conversion of a PAnsiChar UTF-16 buffer into a UTF-8 encoded buffer
- Dest^ buffer must be reserved with at least SourceChars*3 bytes
- will append a trailing #0 to the returned PUTF8Char, unless NoTrailingZero is set

function AnsiToRawUnicode(Source: PAnsiChar; SourceChars: Cardinal): RawUnicode; override;

Convert any UTF-16 Ansi buffer into an Unicode String
- returns a value using our RawUnicode kind of string

function UnicodeBufferToAnsi(Dest: PAnsiChar; Source: PWideChar; SourceChars: Cardinal): PAnsiChar; override;

Direct conversion of an Unicode buffer into a PAnsiChar UTF-16 buffer
- Dest^ buffer must be reserved with at least SourceChars*3 bytes

function UTF8BufferToAnsi(Dest: PAnsiChar; Source: PUTF8Char; SourceChars: Cardinal): PAnsiChar; override;

Direct conversion of an UTF-8 encoded buffer into a PAnsiChar UTF-16 buffer
- Dest^ buffer must be reserved with at least SourceChars bytes
- no trailing #0 is appended to the buffer

TSynTempBuffer = object(TObject)

Implements a stack-based storage of some (UTF-8 or binary) text
- avoid temporary memory allocation via the heap for up to 4KB of data
- could be used e.g. to make a temporary copy when JSON is parsed in-place
- call one of the Init() overloaded methods, then Done to release its memory
- all Init() methods will allocate 16 more bytes, for a trailing #0 and to ensure our fast JSON parsing won’t trigger any GPF (since it may read up to 4 bytes ahead via its PInteger() trick) or any SSE4.2

function Init(SourceLen: PtrInt): pointer; overload;

Initialize a new temporary buffer of a given number of bytes

function Init: integer; overload;

Initialize the buffer returning the internal buffer size (4095 bytes)
- could be used e.g. for an API call, first trying with plain temp.Init and using temp.buf and temp.len safely in the call, only calling temp.Init(expectedsize) if the API returned an error about an insufficient buffer space
function Init(Source: PUTF8Char): PUTF8Char; overload;
   Initialize a temporary copy of the supplied text buffer, ending with #0

function InitIncreasing(Count: PtrInt; Start: PtrInt=0): PIntegerArray;
   Initialize a new temporary buffer filled with 32-bit integer increasing values

function InitOnStack: pointer;
   Initialize a temporary buffer with the length of the internal stack

function InitRandom(RandomLen: integer; forcegsl: boolean=true): pointer;
   Initialize a new temporary buffer of a given number of random bytes
   - will fill the buffer via FillRandom() calls
   - forcegsl is true by default, since Lecuyer's generator has no HW bug

function InitZero(ZeroLen: PtrInt): pointer;
   Initialize a new temporary buffer of a given number of zero bytes

procedure Done; overload;
   Finalize the temporary storage

procedure Done(EndBuf: pointer; var Dest: RawUTF8); overload;
   Finalize the temporary storage, and create a RawUTF8 string from it

procedure Init(Source: pointer; SourceLen: PtrInt); overload;
   Initialize a temporary copy of the supplied text buffer

procedure Init(const Source: RawByteString); overload;
   Initialize a temporary copy of the content supplied as RawByteString
   - will also allocate and copy the ending #0 (even for binary)

TTempUTF8 = record
   A memory structure which avoids a temporary RawUTF8 allocation
   - used by VarRecToTempUTF8() and FormatUTF8()/FormatShort()

TDiv100Rec = packed record
   Small structure used as convenient result to Div100() procedure
   
   D: cardinal;
   Contains V div 100 after Div100(V)

   M: cardinal;
   Contains V mod 100 after Div100(V)

TFindFiles = object(TObject)
   File found result item, as returned by FindFiles()
   - Delphi "object" is buggy on stack -> also defined as record with methods

   Attr: Integer;
   The matching file attributes

   Name: TFileName;
   The matching file name, including its folder name
Size: Int64;
    The matching file size

Timestamp: TDateTime;
    The matching file date/time

function ToText: shortstring;
    Returns some ready-to-be-logged text

procedure FromSearchRec(const Directory: TFileName; const F: TSearchRec);
    Fill the item properties from a FindFirst/FindNext's TSearchRec

TPublishedMethodInfo = record
    Information about one method, as returned by GetPublishedMethods
    Method: TMethod;
        A callback to the method, for the given class instance
    Name: RawUTF8;
        The method name

TMultiPart = record
    Used by MultiPartFormDataDecode() to return one item of its data

TSortedWordArray = object(TObject)
    Used to store and retrieve Words in a sorted array
    - Delphi "object" is buggy on stack -> also defined as record with methods
    Count: PtrInt;
        How many items are currently in Values[]
    Values: TWordDynArray;
        The actual 16-bit word storage
    function Add(aValue: Word): PtrInt;
        Add a value into the sorted array
        - return the index of the new inserted value into the Values[] array
        - return -(foundindex+1) if this value is already in the Values[] array
    function IndexOf(aValue: Word): PtrInt;
        Return the index if the supplied value in the Values[] array
        - return -1 if not found

TSortedIntegerArray = object(TObject)
    Used to store and retrieve Integers in a sorted array
    - Delphi "object" is buggy on stack -> also defined as record with methods
    Count: PtrInt;
        How many items are currently in Values[]
Values: TIntegerDynArray;

*The actual 32-bit integers storage*

**function Add(aValue: integer): PtrInt;**

*Add a value into the sorted array*
- return the index of the new inserted value into the Values[] array
- return -(foundindex+1) if this value is already in the Values[] array

**function IndexOf(aValue: integer): PtrInt;**

*Return the index if the supplied value in the Values[] array*
- return -1 if not found

**TValueResult = record**

*Kind of result returned by FromVarBlob() function*

- **Len: PtrInt;**
  *Value length (in bytes)*

- **Ptr: PAnsiChar;**
  *Start of data value*

**TDynArray = object(TObject)**

*A wrapper around a dynamic array with one dimension*
- provide TList-like methods using fast RTTI information
- can be used to fast save/retrieve all memory content to a TStream
- note that the "const Elem" is not checked at compile time nor runtime: you must ensure that Elem matches the element type of the dynamic array
- can use external Count storage to make Add() and Delete() much faster (avoid most reallocation of the memory buffer)
- Note that TDynArray is just a wrapper around an existing dynamic array: methods can modify the content of the associated variable but the TDynArray doesn't contain any data by itself. It is therefore aimed to initialize a TDynArray wrapper on need, to access any existing dynamic array.
- is defined as an object or as a record, due to a bug in Delphi 2009/2010 compiler (at least): this structure is not initialized if defined as an object on the stack, but will be as a record :(

**function Add(const Elem): PtrInt;**

*Add an element to the dynamic array*
- warning: Elem must be of the same exact type than the dynamic array, and must be a reference to a variable (you can't write Add(i+10) e.g.)
- returns the index of the added element in the dynamic array
- note that because of dynamic array internal memory management, adding may reallocate the list every time a record is added, unless an external count variable has been specified in Init(...,@Count) method
function AddArray(const DynArrayVar; aStartIndex: integer=0; aCount: integer=-1): integer;
  *Add elements from a given dynamic array variable*
  - the supplied source DynArray MUST be of the same exact type as the current used for this TDynArray - warning: pass here a reference to a "array of ..." variable, not another TDynArray instance; if you want to add another TDynArray, use AddDynArray() method
  - you can specify the start index and the number of items to take from the source dynamic array (leave as -1 to add till the end)
  - returns the number of items added to the array

function ClearSafe: boolean;
  *Delete the whole dynamic array content, ignoring exceptions*
  - returns true if no exception occurred when calling Clear, false otherwise
  - you should better not call this method, which will catch and ignore all exceptions - but it may somewhat make sense in a destructor
  - this method will recognize T*ObjArray types and free all instances

function Delete(aIndex: PtrInt): boolean;
  *Delete one item inside the dynamic array*
  - the deleted element is finalized if necessary
  - this method will recognize T*ObjArray types and free all instances

function ElemCopyFirstField(Source, Dest: Pointer): boolean;
  *Will copy the first field value of an array element*
  - will use the array KnownType to guess the copy routine to use
  - returns false if the type information is not enough for a safe copy

function ElemEquals(const A, B): boolean;
  *Compare the content of two elements, returning TRUE if both values equal*
  - this method compares first using any supplied Compare property, then by content using the RTTI element description of the whole record

function ElemLoad(Source: PAnsiChar; SourceMax: PAnsiChar=nil): RawByteString; overload;
  *Load an array element as saved by the ElemSave method*
  - this overloaded method will retrieve the element as a memory buffer, which should be cleared by ElemLoadClear() before release

function ElemLoadFind(Source: PAnsiChar; SourceMax: PAnsiChar=nil): integer;
  *Search for an array element as saved by the ElemSave method*
  - same as ElemLoad() + Find()/IndexOf() + ElemLoadClear()
  - will call Find() method if Compare property is set
  - will call generic IndexOf() method if no Compare property is set

function ElemPtr(index: PtrInt): pointer;
  *Returns a pointer to an element of the array*
  - returns nil if aIndex is out of range
  - since TDynArray is just a wrapper around an existing array, you should better use direct access to its wrapped variable, and not using this slower and more error prone method (such pointer access lacks of strong typing abilities), which was designed for TDynArray internal use
**function** ElemSave(const Elem): RawByteString;

*Save an array element into a serialized binary content*
- use the same layout as TDynArray.SaveTo, but for a single item
- you can use ElemLoad method later to retrieve its content
- warning: Elem must be of the same exact type than the dynamic array, and must be a reference to a variable (you can't write ElemSave(i+10) e.g.)

**function** Equals(const B: TDynArray; ignorecompare: boolean=false): boolean;

*Compare the content of the two arrays, returning TRUE if both match*
- this method compares using any supplied Compare property (unless ignorecompare=true), or by content using the RTTI element description of the whole array items
- will call SaveToJSON to compare T*ObjArray kind of arrays

**function** FastLocateOrAddSorted(const Elem; wasAdded: PBoolean=nil): integer;

*Search and add an element value inside a sorted dynamic array*
- this method will use the Compare property function for the search
- will be faster than a manual FindAndAddIfNotExisting+Sort process
- returns the index of the existing Elem and wasAdded^=false
- returns the sorted index of the inserted Elem and wasAdded^=true
- if the array is not sorted, returns -1 and wasAdded^=false
- is just a wrapper around FastLocateSorted+FastAddSorted

**function** FastLocateSorted(const Elem; out Index: Integer): boolean;

*Search for an element value inside a sorted dynamic array*
- this method will use the Compare property function for the search
- will be faster than a manual FindAndAddIfNotExisting+Sort process
- returns TRUE and the index of existing Elem, or FALSE and the index where the Elem is to be inserted so that the array remains sorted
- you should then call FastAddSorted() later with the returned Index
- if the array is not sorted, returns FALSE and Index=-1
- warning: Elem must be of the same exact type than the dynamic array, and must be a reference to a variable (no FastLocateSorted(i+10) e.g.)

**function** Find(const Elem; const aIndex: TIntegerDynArray; aCompare: TDynArraySortCompare): PtrInt; overload;

*Search for an element value inside the dynamic array, from an external indexed lookup table*
- return the index found (0..Count-1), or -1 if Elem was not found
- this method will use a custom comparison function, with an external integer table, as created by the CreateOrderedIndex() method: it allows multiple search orders in the same dynamic array content
- if an indexed lookup is supplied, it must already be sorted: this function will then use fast O(log(n)) binary search
- if an indexed lookup is not supplied (i.e aIndex=nil), this function will use slower but accurate O(n) iterating search
- warning; the lookup index should be synchronized if array content is modified (in case of adding or deletion)
function Find(const Elem): PtrInt; overload;

*Search for an element value inside the dynamic array*
- this method will use the Compare property function for the search
- return the index found (0..Count-1), or -1 if Elem was not found
- if the array is sorted, it will use fast O(log(n)) binary search
- if the array is not sorted, it will use slower O(n) iterating search
- warning: Elem must be of the same exact type than the dynamic array, and must be a reference to a variable (you can't write Find(i+10) e.g.)

function FindAllSorted(const Elem; out FirstIndex,LastIndex: Integer): boolean;

*Search the elements range which match a given value in a sorted dynamic array*
- this method will use the Compare property function for the search
- returns TRUE and the matching indexes, or FALSE if none found
- if the array is not sorted, returns FALSE

function FindAndAddIfNotExisting(const Elem; aIndex: PIntegerDynArray=nil; aCompare: TDynArraySortCompare=nil): integer;

*Search for an element value, then add it if none matched*
- this method will use the Compare property function for the search, or the supplied indexed lookup table and its associated compare function
- if no Elem content matches, the item will be added to the array
- can be used e.g. as a simple dictionary: if Compare will match e.g. the first string field (i.e. set to SortDynArrayString), you can fill the first string field with the searched value (if returned index is >= 0)
- return the index found (0..Count-1), or -1 if Elem was not found and the supplied element has been successfully added
- if the array is sorted, it will use fast O(log(n)) binary search
- if the array is not sorted, it will use slower O(n) iterating search
- warning: Elem must be of the same exact type than the dynamic array, and must be a reference to a variable (you can't write Find(i+10) e.g.)

function FindAndDelete(const Elem; aIndex: PIntegerDynArray=nil; aCompare: TDynArraySortCompare=nil): integer;

*Search for an element value, then delete it if match*
- this method will use the Compare property function for the search, or the supplied indexed lookup table and its associated compare function
- if Elem content matches, this item will be deleted from the array
- can be used e.g. as a simple dictionary: if Compare will match e.g. the first string field (i.e. set to SortDynArrayString), you can fill the first string field with the searched value (if returned index is >= 0)
- return the index deleted (0..Count-1), or -1 if Elem was not found
- if the array is sorted, it will use fast O(log(n)) binary search
- if the array is not sorted, it will use slower O(n) iterating search
- warning: Elem must be of the same exact type than the dynamic array, and must be a reference to a variable (you can't write Find(i+10) e.g.)
function FindAndFill(var Elem; aIndex: PIntegerDynArray=nil; aCompare: TDynArraySortCompare=nil): integer;

Search for an element value, then fill all properties if match
- this method will use the Compare property function for the search, or the supplied indexed lookup table and its associated compare function
- if Elem content matches, all Elem fields will be filled with the record
- can be used e.g. as a simple dictionary: if Compare will match e.g. the first string field (i.e. set to SortDynArrayString), you can fill the first string field with the searched value (if returned index is >= 0)
- return the index found (0..Count-1), or -1 if Elem was not found
- if the array is sorted, it will use fast O(log(n)) binary search
- if the array is not sorted, it will use slower O(n) iterating search
- warning: Elem must be of the same exact type than the dynamic array, and must be a reference to a variable (you can't write Find(i+10) e.g.)

function FindAndUpdate(const Elem; aIndex: PIntegerDynArray=nil; aCompare: TDynArraySortCompare=nil): integer;

Search for an element value, then update the item if match
- this method will use the Compare property function for the search, or the supplied indexed lookup table and its associated compare function
- if Elem content matches, this item will be updated with the supplied value
- can be used e.g. as a simple dictionary: if Compare will match e.g. the first string field (i.e. set to SortDynArrayString), you can fill the first string field with the searched value (if returned index is >= 0)
- return the index found (0..Count-1), or -1 if Elem was not found
- if the array is sorted, it will use fast O(log(n)) binary search
- if the array is not sorted, it will use slower O(n) iterating search
- warning: Elem must be of the same exact type than the dynamic array, and must be a reference to a variable (you can't write Find(i+10) e.g.)

function GuessKnownType(exactType: boolean=false): TDynArrayKind;

Low-level computation of KnownType and KnownSize fields from RTTI
- do nothing if has already been set at initialization, or already computed

function HasCustomJSONParser: boolean;

Check this dynamic array from the GlobalJSONCustomParsers list
- returns TRUE if this array has a custom JSON parser
function IndexOf(const Elem): PtrInt;

Search for an element value inside the dynamic array
- return the index found (0..Count-1), or -1 if Elem was not found
- will search for all properties content of the element: TList.IndexOf() searches by address, this
  method searches by content using the RTTI element description (and not the Compare property
  function)
- use the Find() method if you want the search via the Compare property function, or e.g. to
  search only with some part of the element content
- will work with simple types: binaries (byte, word, integer, Int64, Currency, array[0..255] of
  byte, packed records with no reference-counted type within...), string types (e.g. array of string),
  and packed records with binary and string types within (like TFileVersion)
- won't work with not packed types (like a shorter string, or a record with byte or word fields with
  {$A+}): in this case, the padding data (i.e. the bytes between the aligned fields can be filled as
  random, and there is no way with standard RTTI do know which they are)
- warning: Elem must be of the same exact type than the dynamic array, and must be a
  reference to a variable (you can't write IndexOf(i+10) e.g.)

function IsVoid: boolean;

Check if the wrapper points to a dynamic array

function LoadFrom(Source: PAnsiChar; AfterEach: TDynArrayAfterLoadFrom=nil;
NoCheckHash: boolean=false; SourceMax: PAnsiChar=nil): PAnsiChar;

Unserialize dynamic array content from binary written by TDynArray.SaveTo
- return nil if the Source buffer is incorrect: invalid type, wrong checksum, or optional
  SourceMax overflow
- return a non nil pointer just after the Source content on success
- this method will raise an ESynException for T*ObjArray types
- you can optionally call AfterEach callback for each row loaded
- if you don't want to allocate all items on memory, but just want to iterate over all items stored
  in a TDynArray.SaveTo memory buffer, consider using TDynArrayLoadFrom object

function LoadFromBinary(const Buffer: RawByteString; NoCheckHash: boolean=false): boolean;

Unserialize the dynamic array content from a TDynArray.SaveTo binary string
- same as LoadFrom, and will check for any buffer overflow since we know the actual end of
  input buffer
function LoadFromJSON(P: PUTF8Char; aEndOfObject: PUTF8Char=nil; CustomVariantOptions: PDocVariantOptions=nil): PUTF8Char;

Load the dynamic array content from an UTF-8 encoded JSON buffer
- expect the format as saved by TTextWriter.AddDynArrayJSON method, i.e. handling
  TBooleanDynArray, TIntIntegerDynArray, TInt64DynArray, TCardinalDynArray, TDoubleDynArray,
  TCurrencyDynArray, TWordDynArray, TByteDynArray, TRawUTF8DynArray, TWinAnsiDynArray,
  TRawByteStringDynArray, TStringDynArray, TWideStringDynArray, TSynUnicodeDynArray,
  TTimeLogDynArray and TDateTimeDynArray as JSON array - or any customized valid JSON
  serialization as set by TTextWriter.RegisterCustomJSONSerializer
- or any other kind of array as Base64 encoded binary stream precessed via
  JSON_BASE64_MAGIC (UTF-8 encoded \uFFF0 special code)
  - typical handled content could be
    ['[1,2,3,4]' or ['\uFFF0base64encodedbinary']]
- return a pointer at the end of the data read from P, nil in case of an invalid input buffer
- this method will recognize T*ObjArray types, and will first free any existing instance before
  unserializing, to avoid memory leak
- warning: the content of P^ will be modified during parsing: please make a local copy if it will be
  needed later (using e.g. TSynTempBufer)

function LoadFromVariant(const DocVariant: variant): boolean;

Load the dynamic array content from a TDocVariant instance
- will convert the TDocVariant into JSON, the call LoadFromJSON

function New: integer;

Add an element to the dynamic array
- this version add a void element to the array, and returns its index
- note: if you use this method to add a new item with a reference to the dynamic array, using a
  local variable is needed under FPC:
    i := DynArray.New;
    with Values[i] do begin // otherwise Values is nil -> GPF
      Field1 := 1;
    end;

function Peek(var Dest): boolean;

Get the last element stored in the dynamic array
- Add + Pop/Peek will implement a LIFO (Last-In-First-Out) stack
- warning: Elem must be of the same exact type than the dynamic array
- returns true if the item was successfully copied into Dest
- use Pop() if you also want to remove the item

function Pop(var Dest): boolean;

Get and remove the last element stored in the dynamic array
- Add + Pop/Peek will implement a LIFO (Last-In-First-Out) stack
- warning: Elem must be of the same exact type than the dynamic array
- returns true if the item was successfully copied and removed
- use Peek() if you don't want to remove the item
function SaveTo: RawByteString; overload;

*Save the dynamic array content into a RawByteString*
- will use a proprietary binary format, with some variable-length encoding of the string length -
  note that if you change the type definition, any previously-serialized content will fail, maybe
  triggering unexpected GPF: use SaveToTypeInfoHash if you share this binary data across
  executables
- this method will raise an ESynException for T*ObjArray types
- use TDynArray.LoadFrom or TDynArrayLoadFrom to decode the saved buffer

function SaveTo(Dest: PAnsiChar): PAnsiChar; overload;

*Save the dynamic array content into an allocated memory buffer*
- Dest buffer must have been allocated to contain at least the number of bytes returned by the
  SaveToLength method
- return a pointer at the end of the data written in Dest, nil in case of an invalid input buffer
- will use a proprietary binary format, with some variable-length encoding of the string length -
  note that if you change the type definition, any previously-serialized content will fail, maybe
  triggering unexpected GPF: use SaveToTypeInfoHash if you share this binary data across
  executables
- this method will raise an ESynException for T*ObjArray types
- use TDynArray.LoadFrom or TDynArrayLoadFrom to decode the saved buffer

function SaveToJSON(EnumSetsAsText: boolean=false; reformat: TTextWriterJSONFormat=jsonCompact): RawUTF8; overload;

*Serialize the dynamic array content as JSON*
- is just a wrapper around TTextWriter.AddDynArrayJSON()
- this method will therefore recognize T*ObjArray types

function SaveToLength: integer;

*Compute the number of bytes needed by SaveTo() to persist a dynamic array*
- will use a proprietary binary format, with some variable-length encoding of the string length -
  note that if you change the type definition, any previously-serialized content will fail, maybe
  triggering unexpected GPF: use SaveToTypeInfoHash if you share this binary data across
  executables
- this method will raise an ESynException for T*ObjArray types

function SaveToTypeInfoHash(crc: cardinal=0): cardinal;

*Compute a crc32c-based hash of the RTTI for this dynamic array*
- can be used to ensure that the TDynArray.SaveTo binary layout is compatible across
  executables
- won't include the RTTI type kind, as TypeInfoToHash(), but only ElemSize or ElemType
  information, or any previously registered TTextWriter.RegisterCustomJSONSerializerFromText
  definition

procedure AddDynArray(const aSource: TDynArray; aStartIndex: integer=0; aCount: integer=-1);

*Add elements from a given TDynArray*
- the supplied source TDynArray MUST be of the same exact type as the current used for this
  TDynArray, otherwise it won't do anything
- you can specify the start index and the number of items to take from the source dynamic array
  (leave as -1 to add till the end)
procedure Clear;

Delete the whole dynamic array content
- this method will recognize T*ObjArray types and free all instances

procedure Copy(const Source: TDynArray; ObjArrayByRef: boolean=false);

Set all content of one dynamic array to the current array
- both must be of the same exact type
- T*ObjArray will be reallocated and copied by content (using a temporary JSON serialization), unless ObjArrayByRef is true and pointers are copied

procedure CopyFrom(const Source; MaxElem: integer; ObjArrayByRef: boolean=false);

Set all content of one dynamic array to the current array
- both must be of the same exact type
- T*ObjArray will be reallocated and copied by content (using a temporary JSON serialization), unless ObjArrayByRef is true and pointers are copied

procedure CopyTo(out Dest; ObjArrayByRef: boolean=false);

Set all content of the current dynamic array to another array variable
- both must be of the same exact type
- resulting length(Dest) will match the exact items count, even if an external Count integer variable is used by this instance
- T*ObjArray will be reallocated and copied by content (using a temporary JSON serialization), unless ObjArrayByRef is true and pointers are copied

procedure CreateOrderedIndex(var aIndex: TIntegerDynArray; aCompare: TDynArraySortCompare); overload;

Sort the dynamic array elements using a lookup array of indexes
- in comparison to the Sort method, this CreateOrderedIndex won't change the dynamic array content, but only create (or update) the supplied integer lookup array, using the specified comparison function
- if aCompare is not supplied, the method will use fCompare (if defined)
- you should provide either a void either a valid lookup table, that is a table with one to one lookup (e.g. created with FillIncreasing)
- if the lookup table has less elements than the main dynamic array, its content will be recreated

procedure CreateOrderedIndex(out aIndex: TSynTempBuffer; aCompare: TDynArraySortCompare); overload;

Sort the dynamic array elements using a lookup array of indexes
- this overloaded method will use the supplied TSynTempBuffer for index storage, so use PIntegerArray(aIndex.buf) to access the values
- caller should always make aIndex.Done once done

procedure CreateOrderedIndexAfterAdd(var aIndex: TIntegerDynArray; aCompare: TDynArraySortCompare);

Sort using a lookup array of indexes, after a Add()
- will resize aIndex if necessary, and set aIndex[Count-1] := Count-1

procedure ElemClear(var Elem);

Will reset the element content

procedure ElemCopy(const A; var B);

Will copy one element content
procedure ElemCopyAt(index: PtrInt; var Dest);

Will copy one element content from its index into another variable
- do nothing if index is out of range

procedure ElemCopyFrom(const Source; index: PtrInt; ClearBeforeCopy: boolean=false);

Will copy one variable content into an indexed element
- do nothing if index is out of range
- ClearBeforeCopy will call ElemClear() before the copy, which may be safer if the source item is a copy of Values[index] with some dynamic arrays

procedure ElemLoad(Source: PAnsiChar; var Elem; SourceMax: PAnsiChar=nil);

Load an array element as saved by the ElemSave method into Elem variable
- warning: Elem must be of the same exact type than the dynamic array, and must be a reference to a variable (you can't write ElemLoad(P,i+10) e.g.)

procedure ElemLoadClear(var ElemTemp: RawByteString);

Finalize a temporary buffer used to store an element via ElemLoad()
- will release any managed type referenced inside the RawByteString, then void the variable
- is just a wrapper around ElemClear(pointer(ElemTemp)) + ElemTemp := ''

procedure ElemMoveTo(index: PtrInt; var Dest);

Will move one element content from its index into another variable
- will erase the internal item after copy
- do nothing if index is out of range

procedure FastAddSorted(Index: Integer; const Elem);

Insert a sorted element value at the proper place
- the index should have been computed by FastLocateSorted(): false
- you may consider using FastLocateOrAddSorted() instead

procedure FastDeleteSorted(Index: Integer);

Delete a sorted element value at the proper place
- plain Delete(Index) would reset the fSorted flag to FALSE, so use this method with a FastLocateSorted/FastAddSorted array
procedure Init(aTypeInfo: pointer; var aValue; aCountPointer: PInteger=nil);

Initialize the wrapper with a one-dimension dynamic array
- the dynamic array must have been defined with its own type (e.g. TIntegerDynArray = array of Integer)
- if aCountPointer is set, it will be used instead of length() to store the dynamic array items count
- it will be much faster when adding elements to the array, because the dynamic array won’t need to be resized each time - but in this case, you should use the Count property instead of length(array) or high(array) when accessing the data: in fact length(array) will store the memory size reserved, not the items count
- if aCountPointer is set, its content will be set to 0, whatever the array length is, or the current aCountPointer^ value is
- a sample usage may be:
  var DA: TDynArray;
  A: TIntegerDynArray;
  begin
    DA.Init(TypeInfo(TIntegerDynArray),A);
    (....)
  end
- a sample usage may be (using a count variable):
  var DA: TDynArray;
  A: TIntegerDynArray;
  ACount: integer;
  i: integer;
  begin
    DA.Init(TypeInfo(TIntegerDynArray),A,@ACount);
    for i := 1 to 100000 do
      DA.Add(i); // MUCH faster using the ACount variable
    (....) // now you should use DA.Count or Count instead of Length(A)

procedure InitFrom(const aAnother: TDynArray; var aValue);

Fast initialize a wrapper for an existing dynamic array of the same type
- is slightly faster than
  Init(aAnother.ArrayType,aValue,nil);

procedure InitSpecific(aTypeInfo: pointer; var aValue; aKind: TDynArrayKind; aCountPointer: PInteger=nil; aCaseInsensitive: boolean=false);

Initialize the wrapper with a one-dimension dynamic array
- this version accepts to specify how comparison should occur, using TDynArrayKind kind of first field
- djNone and djCustom are too vague, and will raise an exception
- no RTTI check is made over the corresponding array layout: you shall ensure that the aKind parameter matches the dynamic array element definition
- aCaseInsensitive will be used for djRawUTF8..djHash512 text comparison

procedure Insert(Index: PInt; const Elem);

Add an element to the dynamic array at the position specified by Index
- warning: Elem must be of the same exact type than the dynamic array, and must be a reference to a variable (you can't write Insert(10,i+10) e.g.)
procedure LoadFromStream(Stream: TCustomMemoryStream);

Load the dynamic array content from a (memory) stream
- stream content must have been created using SaveToStream method
- will handle array of binaries values (byte, word, integer...), array of strings or array of packed records, with binaries and string properties
- will use a proprietary binary format, with some variable-length encoding of the string length
- note that if you change the type definition, any previouslyserialized content will fail, maybe triggering unexpected GPF: use SaveToTypeInfoHash if you share this binary data accross executables

procedure Reverse;

Will reverse all array elements, in place

procedure SaveToJson(out Result: RawUTF8; EnumSetsAsText: boolean=false; reformat: TTextWriterJSONFormat=jsonCompact); overload;

Serialize the dynamic array content as JSON
- is just a wrapper around TTextWriter.AddDynArrayJSON
- this method will therefore recognize T*ObjArray types

procedure SaveToStream(Stream: TStream);

Save the dynamic array content into a (memory) stream
- will handle array of binaries values (byte, word, integer...), array of strings or array of packed records, with binaries and string properties
- will use a proprietary binary format, with some variable-length encoding of the string length
- note that if you change the type definition, any previouslyserialized content will fail, maybe triggering unexpected GPF: use SaveToTypeInfoHash if you share this binary data accross executables
- Stream position will be set just after the added data
- is optimized for memory streams, but will work with any kind of TStream

procedure Slice(var Dest; aCount: Cardinal; aFirstIndex: cardinal=0);

Select a sub-section (slice) of a dynamic array content

procedure Sort(aCompare: TDynArraySortCompare=nil); overload;

Sort the dynamic array elements, using the Compare property function
- it will change the dynamic array content, and exchange all elements in order to be sorted in increasing order according to Compare function

procedure Sort(const aCompare: TEventDynArraySortCompare; aReverse: boolean=false); overload;

Sort the dynamic array elements, using a Compare method (not function)
- it will change the dynamic array content, and exchange all elements in order to be sorted in increasing order according to Compare function, unless aReverse is true
- it won't mark the array as Sorted, since the comparer is local

procedure SortRange(aStart, aStop: integer; aCompare: TDynArraySortCompare=nil);

Sort some dynamic array elements, using the Compare property function
- this method allows to sort only some part of the items
- it will change the dynamic array content, and exchange all elements in order to be sorted in increasing order according to Compare function
procedure UseExternalCount(var aCountPointer: Integer);
Define the reference to an external count integer variable
- Init and InitSpecific methods will reset the aCountPointer to 0: you can use this method to set
the external count variable without overriding the current value

procedure Void;
Initialize the wrapper to point to no dynamic array

property ArrayType: pointer read fTypeInfo;
The known RTTI information of the whole array

property ArrayTypeName: RawUTF8 read GetArrayTypeName;
The known type name of the whole array, as RawUTF8

property ArrayTypeShort: PShortString read GetArrayTypeName;
The known type name of the whole array, as PShortString

property Capacity: PtrInt read GetCapacity write SetCapacity;
The internal buffer capacity
- if no external Count pointer was set with Init, is the same as Count
- if an external Count pointer is set, you can set a value to this property before a massive use of
  the Add() method e.g.
- if no external Count pointer is set, set a value to this property will affect the Count value, i.e.
  Add() will append after this count
- this property will recognize T*ObjArray types, so will free any stored instance if the array is
  sized down

property Compare: TDynArraySortCompare read fCompare write SetCompare;
The compare function to be used for Sort and Find methods
- by default, no comparison function is set
- common functions exist for base types: e.g. SortDynArrayByte, SortDynArrayBoolean,
  SortDynArrayWord, SortDynArrayInteger, SortDynArrayCardinal, SortDynArraySingle,
  SortDynArrayInt64, SortDynArrayDouble, SortDynArrayAnsiString, SortDynArrayAnsiStringI,
  SortDynArrayString, SortDynArrayStringI, SortDynArrayUnicodeString,
  SortDynArrayUnicodeStringI

property Count: PtrInt read GetCount write SetCount;
Retrieve or set the number of elements of the dynamic array
- same as length(DynArray) or SetLength(DynArray)
- this property will recognize T*ObjArray types, so will free any stored instance if the array is
  sized down

property ElemSize: cardinal read fElemSize;
The internal in-memory size of one element, as retrieved from RTTI

property ElemType: pointer read fElemType;
The internal type information of one element, as retrieved from RTTI

property IsObjArray: boolean read GetIsObjArray write SetIsObjArray;
If this dynamic array is a T*ObjArray

property KnownSize: integer read fKnownSize;
The raw storage size of the first field KnownType
property KnownType: TDynArrayKind read fKnownType;

The first field recognized type
- could have been set at initialization, or after a GuessKnownType call

property Sorted: boolean read fSorted write fSorted;

Must be TRUE if the array is currently in sorted order according to the compare function
- Add/Delete/Insert/Load* methods will reset this property to false
- Sort method will set this property to true
- you MUST set this property to false if you modify the dynamic array content in your code, so that Find() won't try to wrongly use binary search in an unsorted array, and miss its purpose

property Value: PPointer read fValue;

Low-level direct access to the storage variable

TDynArrayLoadFrom = object(TObject)

Allows to iterate over a TDynArray.SaveTo binary buffer
- may be used as alternative to TDynArray.LoadFrom, if you don't want to allocate all items at once, but retrieve items one by one

Count: integer;

How many items were saved in the TDynArray.SaveTo binary buffer
- equals -1 if Init() failed to unserialize its header

Current: integer;

The zero-based index of the current item pointed by next Step() call
- is in range 0..Count-1 until Step() returns false

Position: PAnsiChar;

Current position in the TDynArray.SaveTo binary buffer
- after Step() returned false, points just after the binary buffer, like a regular TDynArray.LoadFrom

function CheckHash: boolean;

After all items are read by Step(), validate the stored hash
- returns true if items hash is correct, false otherwise

function FirstField(out Field): boolean;

Extract the first field value of the current stored item
- returns true if Field was filled with one value, or false if all items were read, and Position contains the end of the binary buffer
- could be called before Step(), to pre-allocate a new item instance, or update an existing instance

function Init(ArrayTypeInfo: pointer; Source: PAnsiChar; SourceMaxLen: PtrInt=0): boolean; overload;

Initialize iteration over a TDynArray.SaveTo binary buffer
- returns true on success, with Count and Position being set
- returns false if the supplied binary buffer is not correct
- you can specify an optional SourceMaxLen to avoid any buffer overflow
function Init(ArrayTypeInfo: pointer; const Source: RawByteString): boolean; overload;

Initialize iteration over a TDynArray.SaveTo binary buffer
- returns true on success, with Count and Position being set
- returns false if the supplied binary buffer is not correct

function Step(out Elem): boolean;

Iterate over the current stored item
- Elem should point to a variable of the exact item type stored in this dynamic array
- returns true if Elem was filled with one value, or false if all items were read, and Position contains the end of the binary buffer

TDynArrayHasher = object(TObject)

Allow O(1) lookup to any dynamic array content
- this won't handle the storage process (like add/update), just efficiently maintain a hash table over an existing dynamic array: several TDynArrayHasher could be applied to a single TDynArray wrapper
- TDynArrayHashed will use a TDynArrayHasher for its own store

Compare: TDynArraySortCompare;

Associated item comparison - may differ from DynArray^.Compare

CountTrigger: integer;

After how many FindBeforeAdd() or Scan() the hashing starts - default 32

EventCompare: TEventDynArraySortCompare;

Custom method-based comparison function

function Find(Elem: pointer): integer; overload;

Search for an element value inside the dynamic array with hashing

function Find(Elem: pointer; aHashCode: cardinal): integer; overload;

Search for a hashed element value inside the dynamic array with hashing

function Find(aHashCode: cardinal; aForAdd: boolean): integer; overload;

Search for a hash position inside the dynamic array with hashing

function FindBeforeAdd(Elem: pointer; out wasAdded: boolean; aHashCode: cardinal): integer;

Search an hashed element value for adding, updating the internal hash table
- trigger hashing if Count reaches CountTrigger

function FindBeforeDelete(Elem: pointer): integer;

Search and delete an element value, updating the internal hash table

function FindOrNew(aHashCode: cardinal; Elem: pointer; aHashTableIndex: PInteger=nil): integer;

Returns position in array, or next void index in HashTable[] as -(index+1)

function GetHashFromIndex(aIndex: PInt): cardinal;

Retrieve the low-level hash of a given item
function HashOne(Elem: pointer): cardinal;
   Compute the hash of a given item

function ReHash(forced, forceGrow: boolean): integer;
   Full computation of the internal hash table
   - returns the number of duplicated values found

function Scan(Elem: pointer): integer;
   Search for an element value inside the dynamic array without hashing
   - trigger hashing if ScanCounter reaches CountTrigger*2

procedure Clear;
   Reset the hash table - no rehash yet

procedure Init(aDynArray: PDynArray; aHashElement: TDynArrayHashOne; aEventHash: TEventDynArrayHashOne; aHasher: THasher; aCompare: TDynArraySortCompare; aEventCompare: TEventDynArraySortCompare; aCaseInsensitive: boolean);
   Initialize the hash table for a given dynamic array storage
   - you can call this method several times, e.g. if aCaseInsensitive changed

procedure InitSpecific(aDynArray: PDynArray; aKind: TDynArrayKind; aCaseInsensitive: boolean);
   Initialize a known hash table for a given dynamic array storage
   - you can call this method several times, e.g. if aCaseInsensitive changed

procedure SetEventHash(const event: TEventDynArrayHashOne);
   Allow custom hashing via a method event

TDynArrayHashed = object(TDynArray)
   Used to access any dynamic array elements using fast hash
   - by default, binary sort could be used for searching items for TDynArray: using a hash is faster on huge arrays for implementing a dictionary
   - in this current implementation, modification (update or delete) of an element is not handled yet: you should rehash all content - only TDynArrayHashed.FindHashedForAdding / FindHashedAndUpdate / FindHashedAndDelete will refresh the internal hash
   - this object extends the TDynArray type, since presence of Hashs[] dynamic array will increase code size if using TDynArrayHashed instead of TDynArray
   - in order to have the better performance, you should use an external Count variable, AND set the Capacity property to the expected maximum count (this will avoid most ReHash calls for FindHashedForAdding+FindHashedAndUpdate)

function AddAndMakeUniqueName(aName: RawUTF8): pointer;
   Search for a given element name, make it unique, and add it to the array
   - expected element layout is to have a RawUTF8 field at first position
   - the aName is searched (using hashing) to be unique, and if not the case, some suffix is added to make it unique
   - use internally FindHashedForAdding method
   - this version will set the field content with the unique value
   - returns a pointer to the newly added element (to set other fields)
function AddUniqueName(const aName: RawUTF8; aNewIndex: PInteger=nil): pointer; overload;

Ensure a given element name is unique, then add it to the array
- just a wrapper to AddUniqueName(aName,"",[],aNewIndex)

function AddUniqueName(const aName: RawUTF8; const ExceptionMsg: RawUTF8; const ExceptionArgs: array of const; aNewIndex: PInteger=nil): pointer; overload;

Ensure a given element name is unique, then add it to the array
- expected element layout is to have a RawUTF8 field at first position
- the aName is searched (using hashing) to be unique, and if not the case, an ESynException.CreateUTF8() is raised with the supplied arguments
- use internally FindHashedForAdding method
- this version will set the field content with the unique value
- returns a pointer to the newly added element (to set other fields)

function FindFromHash(const Elem; aHashCode: cardinal): integer;

Search for an element value inside the dynamic array using its hash
- returns -1 if not found, or the index in the dynamic array if found
- aHashCode parameter contains an already hashed value of the item, to be used e.g. after a call to HashFind()

function FindHashed(const Elem): integer;

Search for an element value inside the dynamic array using hashing
- Elem should be of the type expected by both the hash function and Equals/Compare methods: e.g. if the searchedhashed field in a record is a string as first field, you can safely use a string variable as Elem
- Elem must refer to a variable: e.g. you can’t write FindHashed(i+10)
- will call fHashElement(Elem,fHasher) to compute the needed hash
- returns -1 if not found, or the index in the dynamic array if found

function FindHashedAndDelete(const Elem; FillDeleted: pointer=nil; noDeleteEntry: boolean=false): integer;

Search for an element value inside the dynamic array using hashing, and delete it if matches
- return the index deleted (0..Count-1), or -1 if Elem was not found
- can optionally copy the deleted item to FillDeleted^ before erased
- Elem should be of the type expected by both the hash function and Equals/Compare methods, and must refer to a variable: e.g. you can’t write FindHashedAndDelete(i+10)
- it won’t call slow ReHash but refresh the hash table as needed

function FindHashedAndFill(var ElemToFill): integer;

Search for an element value inside the dynamic array using hashing, and fill Elem with the found content
- return the index found (0..Count-1), or -1 if Elem was not found
- ElemToFill should be of the type expected by the dynamic array, since all its fields will be set on match
function FindHashedAndUpdate(const Elem; AddIfNotExisting: boolean): integer;

Search for an element value inside the dynamic array using hashing, then update any matching item, or add the item if none matched
- by design, hashed field shouldn't have been modified by this update, otherwise the method won't be able to find and update the old hash: in this case, you should first call
  FindHashedAndDelete(OldElem) then FindHashedForAdding(NewElem) to properly handle the internal hash table
- if AddIfNotExisting is FALSE, returns the index found (0..Count-1), or -1 if Elem was not found - update will force slow rehash all content
- if AddIfNotExisting is TRUE, returns the index found (0..Count-1), or the index newly created/added is the Elem value was not matching - add won't rehash all content - for even faster process (avoid ReHash), please set the Capacity property
- Elem should be of the type expected by the dynamic array, since its content will be copied into the dynamic array, and it must refer to a variable: e.g. you can't write
  FindHashedAndUpdate(i+10)

function FindHashedForAdding(const Elem; out wasAdded: boolean; noAddEntry: boolean=false): integer; overload;

Search for an element value inside the dynamic array using hashing, and add a void entry to the array if was not found (unless noAddEntry is set)
- this method will use hashing for fast retrieval
- Elem should be of the type expected by both the hash function and Equals/Compare methods: e.g. if the searched/hashed field in a record is a string as first field, you can safely use a string variable as Elem
- returns either the index in the dynamic array if found (and set wasAdded to false), either the newly created index in the dynamic array (and set wasAdded to true)
- for faster process (avoid ReHash), please set the Capacity property
- warning: in contrast to the Add() method, if an entry is added to the array (wasAdded=true), the entry is left VOID: you must set the field content to expecting value - in short, Elem is used only for searching, not copied to the newly created entry in the array - check
  FindHashedAndUpdate() for a method actually copying Elem fields

function FindHashedForAdding(const Elem; out wasAdded: boolean; aHashCode: cardinal; noAddEntry: boolean=false): integer; overload;

Search for an element value inside the dynamic array using hashing, and add a void entry to the array if was not found (unless noAddEntry is set)
- overloaded method accepting an already hashed value of the item, to be used e.g. after a call to HashFind()

function ReHash(forAdd: boolean=false; forceGrow: boolean=false): integer;

Will compute all hash from the current elements of the dynamic array
- is called within the TDynArrayHashed.Init method to initialize the internal hash array
- can be called on purpose, when modifications have been performed on the dynamic array content (e.g. in case of element deletion or update, or after calling LoadFrom/Clear method) - this is not necessary after FindHashedForAdding / FindHashedAndUpdate / FindHashedAndDelete methods
- returns the number of duplicated items found - which won't be available by hashed
FindHashed() by definition
function Scan(const Elem): integer;

Will search for an element value inside the dynamic array without hashing
- is used internally when Count < HashCountTrigger
- is preferred to Find(), since EventCompare would be used if defined
- Elem should be of the type expected by both the hash function and Equals/Compare methods, and must refer to a variable: e.g. you can't write Scan(i+10)
- returns -1 if not found, or the index in the dynamic array if found
- an internal algorithm can switch to hashing if Scan() is called often, even if the number of items is lower than HashCountTrigger

procedure Init(aTypeInfo: pointer; var aValue; aHashElement: TDynArrayHashOne=nil; aCompare: TDynArraySortCompare=nil; aHasher: THasher=nil; aCountPointer: PInteger=nil; aCaseInsensitive: boolean=false);

Initialize the wrapper with a one-dimension dynamic array
- this version accepts some hash-dedicated parameters: aHashElement to set how to hash each element, aCompare to handle hash collision
- if no aHashElement is supplied, it will hash according to the RTTI, i.e. strings or binary types, and the first field for records (strings included)
- if no aCompare is supplied, it will use default Equals() method
- if no THasher function is supplied, it will use the one supplied in DefaultHasher global variable, set to crc32c() by default - using SSE4.2 instruction if available
- if aCaseInsensitive is set to TRUE, it will ignore difference in 7 bit alphabetic characters (e.g. compare 'a' and 'A' as equal)

procedure InitSpecific(aTypeInfo: pointer; var aValue; aKind: TDynArrayKind; aCountPointer: PInteger=nil; aCaseInsensitive: boolean=false);

Initialize the wrapper with a one-dimension dynamic array
- this version accepts to specify how both hashing and comparison should occur, setting the TDynArrayKind kind of first/hashed field
- djNone and djCustom are too vague, and will raise an exception
- no RTTI check is made over the corresponding array layout: you shall ensure that aKind matches the dynamic array element definition
- aCaseInsensitive will be used for djRawUTF8..djHash512 text comparison

property EventCompare: TEventDynArraySortCompare read fHash.EventCompare write fHash.EventCompare;

Alternative event-oriented Compare function to be used for Sort and Find
- will be used instead of Compare, to allow object-oriented callbacks

property EventHash: TEventDynArrayHashOne read fHash.EventHash write SetEventHash;

Alternative event-oriented Hash function for ReHash
- this object-oriented callback will be used instead of HashElement on each dynamic array entries - HashElement will still be used on const Elem values, since they may be just a sub part of the stored entry

property Hash[aIndex: PtrInt]: Cardinal read GetHashFromIndex;

Retrieve the hash value of a given item, from its index
property HashCountTrigger: integer read fHash.CountTrigger write fHash.CountTrigger;

After how many items the hashing take place
- for smallest arrays, O(n) search if faster than O(1) hashing, since maintaining internal hash table has some CPU and memory costs
- internal search is able to switch to hashing if it founds out that it may have some benefit, e.g. if Scan() is called 2*HashCountTrigger times
- equals 32 by default, i.e. start hashing when Count reaches 32 or manual Scan() is called 64 times

property HashElement: TDynArrayHashOne read fHash.HashElement;

Custom hash function to be used for hashing of a dynamic array element

property Hasher: TDynArrayHasher read fHash;
Access to the internal hash table
- you can call e.g. Hasher.Clear to invalidate the whole hash table

IObjectDynArray = interface(IInterface)

Defines a wrapper interface around a dynamic array of TObject
- implemented by TObjectDynArrayWrapper for instance
- i.e. most common methods are available to work with a dynamic array
- warning: the IObjectDynArray MUST be defined in the stack, class or record BEFORE the dynamic array it is wrapping, otherwise you may leak memory - see for instance TSQLRestServer class:
  fSessionAuthentications: IObjectDynArray; // defined before the array
  fSessionAuthentication: TSQLRestServerAuthenticationDynArray;
note that allocation time as variable on the local stack may depend on the compiler, and its optimization

function Add(Instance: TObject): integer;
Add one element to the dynamic array of TObject instances
- once added, the Instance will be owned by this TObjectDynArray instance

function Capacity: integer;
Returns the internal array capacity of TObject instances available
- which is in fact the length() of the associated dynamic array

function Count: integer;
Returns the number of TObject instances available
- note that the length of the associated dynamic array is used to store the capacity of the list, so won’t probably never match with this value

function Find(Instance: TObject): integer;
Search one element within the TObject instances

procedure Clear;
Delete all TObject instances, and release the memory
- is not to be called for most use, thanks to reference-counting memory handling, but can be handy for quick release

procedure Delete(Index: integer);
Delete one element from the TObject dynamic array
- deleted TObject instance will be freed as expected
**procedure** Slice;

*Ensure the internal list capacity is set to the current Count*
- may be used to publish the associated dynamic array with the expected final size, once
  IObjectDynArray is out of scope

**procedure** Sort(Compare: TDynArraySortCompare);

*Sort the dynamic array content according to a specified comparer*

---

** TObjectDynArrayWrapper = class(TInterfacedObject)**

*A wrapper to own a dynamic array of TObject*
- this version behave list a TObjectList (i.e. owning the class instances)
- but the dynamic array is NOT owned by the instance
- will define an internal Count property, using the dynamic array length as capacity: adding and
deleting will be much faster
- implements IObjectDynArray, so that most common methods are available to work with the
  dynamic array
- does not need any sub-classing of generic overhead to work, and will be reference counted
- warning: the IObjectDynArray MUST be defined in the stack, class or record BEFORE the dynamic
  array it is wrapping, otherwise you may leak memory, and TObjectDynArrayWrapper.Destroy will
  raise an ESynException
- warning: issues with Delphi 10.4 Sydney were reported, which seemed to change the order of
  fields finalization, so the whole purpose of this wrapper may have become incompatible with
  Delphi 10.4 and up
- a sample usage may be:

```pascal
var DA: IObjectDynArray; // defined BEFORE the dynamic array itself
  A: array of TMyObject;
  i: integer;
begin
  DA := TObjectDynArrayWrapper.Create(A);
  DA.Add(TMyObject.Create('one'));
  DA.Add(TMyObject.Create('two'));
  DA.Delete(0);
  assert(DA.Count=1);
  assert(A[0].Name='two');
  DA.Clear;
  assert(DA.Count=0);
  DA.Add(TMyObject.Create('new'));
  assert(DA.Count=1);
end; // will auto-release DA (no need of try..finally DA.Free)
```

**constructor** Create(var aValue; aOwnObjects: boolean=true);

*Initialize the wrapper with a one-dimension dynamic array of TObject*
- by default, objects will be owned by this class, but you may set aOwnObjects=false if you
  expect the dynamic array to remain available

**destructor** Destroy; override;

*Will release all associated TObject instances*

**function** Add(Instance: TObject): integer;

*Add one element to the dynamic array of TObject instances*
- once added, the Instance will be owned by this TObjectDynArray instance (unless aOwnObjects
  was false in Create)*
function Capacity: integer;
    Returns the internal array capacity of TObject instances available
    - which is in fact the length() of the associated dynamic array

function Count: integer;
    Returns the number of TObject instances available
    - note that the length() of the associated dynamic array is used to store the capacity of the list,
      so won't probably never match with this value

function Find(Instance: TObject): integer;
    Search one element within the TObject instances

procedure Clear;
    Delete all TObject instances, and release the memory
    - is not to be called for most use, thanks to reference-counting memory handling, but can be
      handy for quick release
    - warning: won't release the instances if aOwnObjects was false in Create

procedure Delete(Index: integer);
    Delete one element from the TObject dynamic array
    - deleted TObject instance will be freed as expected (unless aOwnObjects was defined as false in Create)

procedure Slice;
    Ensure the internal list capacity is set to the current Count
    - may be used to publish the associated dynamic array with the expected final size, once
      TObjectDynArray is out of scope

procedure Sort(Compare: TDynArraySortCompare);
    Sort the dynamic array content according to a specified comparer

TPersistentWithCustomCreate = class(TPersistent)
    Abstract parent class with a virtual constructor, ready to be overridden to initialize the instance
    - you can specify such a class if you need an object including published properties (like TPersistent)
      with a virtual constructor (e.g. to initialize some nested class properties)

constructor Create; virtual;
    This virtual constructor will be called at instance creation
    - this constructor does nothing, but is declared as virtual so that inherited classes may safely
      override this default void implementation

TInterfacedObjectWithCustomCreate = class(TInterfacedObject)
    Abstract parent class with threadsafe implementation of IInterface and a virtual constructor
    - you can specify e.g. such a class to TSQLRestServer.ServiceRegister() if you need an interfaced
      object with a virtual constructor, ready to be overridden to initialize the instance

constructor Create; virtual;
    This virtual constructor will be called at instance creation
    - this constructor does nothing, but is declared as virtual so that inherited classes may safely
      override this default void implementation
procedure RefCountUpdate(Release: boolean); virtual;

*Used to mimic TInterfacedObject reference counting*
- Release=true will call TInterfacedObject._Release
- Release=false will call TInterfacedObject._AddRef
- could be used to emulate proper reference counting of the instance via interfaces variables, but still storing plain class instances (e.g. in a global list of instances)

**TSynPersistent = class(TObject)**

*Our own empowered TPersistent-like parent class*
- TPersistent has an unexpected speed overhead due a giant lock introduced to manage property name fixup resolution (which we won't use outside the VCL)
- this class has a virtual constructor, so is a preferred alternative to both TPersistent and TPersistentWithCustomCreate classes
- for best performance, any type inheriting from this class will bypass some regular steps: do not implement interfaces or use TMonitor with them!

**constructor Create; virtual;**

*This virtual constructor will be called at instance creation*
- this constructor does nothing, but is declared as virtual so that inherited classes may safely override this default void implementation

**class function NewInstance: TObject; override;**

*Optimized initialization code*
- somewhat faster than the regular RTL implementation - especially since rewritten in pure asm on Delphi/x86
- warning: this optimized version won't initialize the vmtIntfTable for this class hierarchy: as a result, you would NOT be able to implement an interface with a TSynPersistent descendent (but you should not need to, but inherit from TInterfacedObject)
- warning: under FPC, it won't initialize fields management operators

**procedure Assign(Source: TSynPersistent); virtual;**

*Allows to implement a TPersistent-like assignement mechanism*
- inherited class should override AssignTo() protected method to implement the proper assignment

**procedure FreeInstance; override;**

*Optimized x86 asm finalization code*
- warning: this version won't release either any allocated TMonitor (as available since Delphi 2009) - do not use TMonitor with TSynPersistent, but rather the faster TSynPersistentLock class

**TSynList = class(TSynPersistent)**

*Simple and efficient TList, without any notification*
- regular TList has an internal notification mechanism which slows down basic process, and most used methods were not defined as virtual, so can't be easily inherited
- stateless methods (like Add/Clear/Exists/Remove) are defined as virtual since can be overridden e.g. by TSynObjectListLocked to add a TSynLocker

**function Add(item: pointer): integer; virtual;**

*Add one item to the list*
**function** Exists(item: pointer): boolean; *virtual;*
Fast check if one item exists in the list

**function** IndexOf(item: pointer): integer; *virtual;*
Fast retrieve one item in the list

**function** Remove(item: pointer): integer; *virtual;*
Fast delete one item in the list

**procedure** Clear; *virtual;*
Delete all items of the list

**procedure** Delete(index: integer); *virtual;*
Delete one item from the list

**property** Count: integer *read* fCount;
How many items are stored in this TList instance

**property** Items[index: Integer]: pointer *read* Get;
Low-level array-like access to the items stored in this TList instance
- warning: if index is out of range, will return nil and won’t raise any exception

**property** List: TPointerDynArray *read* fList;
Low-level access to the items stored in this TList instance

TSynObjectList = **class**(TSynList)
Simple and efficient TObjectList, without any notification

**constructor** Create(aOwnObjects: boolean=true); *reintroduce;*
Initialize the object list

**destructor** Destroy; *override;*
Finalize the store items

**procedure** Clear; *override;*
Delete all objects of the list

**procedure** ClearFromLast; *virtual;*
Delete all objects of the list in reverse order
- for some kind of processes, owned objects should be removed from the last added to the first

**procedure** Delete(index: integer); *override;*
Delete one object from the list
TSynLocker = object(TObject)

Allow to add cross-platform locking methods to any class instance
- typical use is to define a Safe: TSynLocker property, call Safe.Init and Safe.Done in
constructor/destructor methods, and use Safe.Lock/UnLock methods in a try ... finally section
- in respect to the TCriticalSection class, fix a potential CPU cache line conflict which may degrade
the multi-threading performance, as reported by
@http://www.delphitools.info/2011/11/30/fixing-tcriticalsection
- internal padding is used to safely store up to 7 values protected from concurrent access with a
mutex, so that SizeOf(TSynLocker)>128
- for object-level locking, see TSynPersistentLock which owns one such instance, or call low-level
fSafe := NewSynLocker in your constructor, then fSafe^..DoneAndFreemem in your destructor

Padding: array[0..6] of TVarData;
Internal padding data, also used to store up to 7 variant values
- this memory buffer will ensure no CPU cache line mixup occurs
- you should not use this field directly, but rather the Locked[], LockedInt64[], LockedUTF8[] or
LockedPointer[] methods
- if you want to access those array values, ensure you protect them using a Safe.Lock; try ...
Padding[n] ... finally Safe.Unlock structure, and maintain the PaddingUsedCount field accurately

PaddingUsedCount: integer;
Number of values stored in the internal Padding[] array
- equals 0 if no value is actually stored, or a 1..7 number otherwise
- you should not have to use this field, but for optimized low-level direct access to Padding[]
values, within a Lock/UnLock safe block

function LockedExchange(Index: integer; const Value: variant): variant;
Safe locked in-place exchange of a Variant value
- you may store up to 7 variables, using an 0..6 index, shared with Locked and LockedUTF8 array
properties
- returns the previous stored value, or null if the Index is out of range

function LockedInt64Increment(Index: integer; const Increment: Int64): Int64;
Safe locked in-place increment to an Int64 value
- you may store up to 7 variables, using an 0..6 index, shared with Locked and LockedUTF8 array
properties
- Int64s will be stored internally as a varInt64 variant
- returns the newly stored value
- if the internal value is not defined yet, would use 0 as default value

function LockedPointerExchange(Index: integer; Value: pointer): pointer;
Safe locked in-place exchange of a pointer/TObject value
- you may store up to 7 variables, using an 0..6 index, shared with Locked and LockedUTF8 array
properties
- pointers will be stored internally as a varUnknown variant
- returns the previous stored value, nil if the Index is out of range, or does not store a pointer
function ProtectMethod: IUnknown;

Will enter the mutex until the IUnknown reference is released
- could be used as such under Delphi:
begin
  ... // unsafe code
  Safe.ProtectMethod;
  ... // thread-safe code
end; // Local hidden IUnknown will release the lock for the method

- warning: under FPC, you should assign its result to a local variable – see bug
http://bugs.freepascal.org/view.php?id=26602
var
  LockFPC: IUnknown;
begin
  ... // unsafe code
  LockFPC := Safe.ProtectMethod;
  ... // thread-safe code
end; // Local hidden IUnknown will release the lock for the method

or
begin
  ... // unsafe code
  with Safe.ProtectMethod do begin
    ... // thread-safe code
    end; // Local hidden IUnknown will release the lock for the method
end;

function TryLock: boolean;

Will try to acquire the mutex
- use as such to avoid race condition (from a Safe: TSynLocker property):
if Safe.TryLock then
  try
    ...
  finally
    Safe.Unlock;
  end;

function TryLockMS(retryms: integer): boolean;

Will try to acquire the mutex for a given time
- use as such to avoid race condition (from a Safe: TSynLocker property):
if Safe.TryLockMS(100) then
  try
    ...
  finally
    Safe.Unlock;
  end;

procedure Done;

Finalize the mutex
- calling this method is mandatory (e.g. in the class destructor owning the TSynLocker instance),
otherwise you may encounter unexpected behavior, like access violations or memory leaks

procedure DoneAndFreeMem;

Finalize the mutex, and call FreeMem() on the pointer of this instance
- should have been initiaized with a NewSynLocker call
procedure Init;
  Initialize the mutex
  - calling this method is mandatory (e.g. in the class constructor owning the TSynLocker instance),
    otherwise you may encounter unexpected behavior, like access violations or memory leaks

procedure Lock;
  Lock the instance for exclusive access
  - this method is re-entrant from the same thread (you can nest Lock/UnLock calls in the same
    thread), but would block any other Lock attempt in another thread
  - use as such to avoid race condition (from a Safe: TSyn Locker property):
    Safe.Lock;
    try
      ...
    finally
      Safe.Unlock;
    end;

procedure UnLock;
  Release the instance for exclusive access
  - each Lock/TryLock should have its exact UnLock opposite, so a try..finally block is mandatory
    for safe code

property IsInitialized: boolean read fInitialized;
  Returns true if the Init method has been called for this mutex
  - is only relevant if the whole object has been previously filled with 0, i.e. as part of a class or as
    global variable, but won't be accurate when allocated on stack

property IsLocked: boolean read fLocked;
  Returns true if the mutex is currently locked by another thread

property Locked[Index: integer]: Variant read GetVariant write SetVariant;
  Safe locked access to a Variant value
  - you may store up to 7 variables, using an 0..6 index, shared with Locked Bool, Locked Int64,
    Locked Pointer and Locked UTF8 array properties
  - returns null if the Index is out of range

property LockedBool[Index: integer]: boolean read GetBool write SetBool;
  Safe locked access to a boolean value
  - you may store up to 7 variables, using an 0..6 index, shared with Locked, Locked Int64,
    Locked Pointer and Locked UTF8 array properties
  - value will be stored internally as a var Boolean variant
  - returns nil if the Index is out of range, or does not store a boolean

property LockedInt64[Index: integer]: Int64 read GetInt64 write SetInt64;
  Safe locked access to a Int64 value
  - you may store up to 7 variables, using an 0..6 index, shared with Locked and Locked UTF8 array
    properties
  - Int64s will be stored internally as a var Int64 variant
  - returns nil if the Index is out of range, or does not store a Int64
property LockedPointer[Index: integer]: Pointer read GetPointer write SetPointer;
    Safe locked access to a pointer/TObject value
    - you may store up to 7 variables, using an 0..6 index, shared with Locked, LockedBool,
      LockedInt64 and LockedUTF8 array properties
    - pointers will be stored internally as a varUnknown variant
    - returns nil if the Index is out of range, or does not store a pointer

property LockedUTF8[Index: integer]: RawUTF8 read GetUTF8 write SetUTF8;
    Safe locked access to an UTF-8 string value
    - you may store up to 7 variables, using an 0..6 index, shared with Locked and LockedPointer
      array properties
    - UTF-8 string will be stored internally as a varString variant
    - returns "" if the Index is out of range, or does not store a string

property UnlockedInt64[Index: integer]: Int64 read GetUnlockedInt64 write SetUnlockedInt64;
    Unsafe access to a Int64 value
    - you may store up to 7 variables, using an 0..6 index, shared with Locked and LockedUTF8 array
      properties
    - Int64s will be stored internally as a varInt64 variant
    - returns nil if the Index is out of range, or does not store a Int64
    - you should rather call LockedInt64[] property, or use this property with a Lock; try ... finally
      UnLock block

TSynPersistentLock = class(TSynPersistent)
    Adding locking methods to a TSynPersistent with virtual constructor
    - you may use this class instead of the RTL TCriticalSection, since it would use a TSynLocker which
      does not suffer from CPU cache line conflict

constructor Create; override;
    TSynLocker would increase inherited fields offset initialize the instance, and its associated lock

destructor Destroy; override;
    Finalize the instance, and its associated lock

property Safe: PSynLocker read fSafe;
    Access to the associated instance critical section
    - call Safe.Lock/UnLock to protect multi-thread access on this storage

TSynPersistentLocked = class(TSynPersistentLock)
    Used for backward compatibility only with existing code

TInterfacedObjectLocked = class(TInterfacedObjectWithCustomCreate)
    Adding locking methods to a TInterfacedObject with virtual constructor

constructor Create; override;
    TSynLocker would increase inherited fields offset initialize the object instance, and its associated
    lock
**destructor** Destroy; **override**;

Release the instance (including the locking resource)

**property** Safe: PSynLocker **read** fSafe;

Access to the locking methods of this instance
- use Safe.Lock/TryLock with a try ... finally Safe.Unlock block

---

**TRawUTF8InterningSlot** = **object**(TObject)

Used to store one list of hashed RawUTF8 in TRawUTF8Interning pool
- Delphi "object" is buggy on stack -> also defined as record with methods

Safe: TSynLocker;

Associated mutex for thread-safe process

Value: TRawUTF8DynArray;

Actual RawUTF8 storage

Values: TDynArrayHashed;

Hashed access to the Value[] list

**function** Clean(aMaxRefCount: integer): integer;

Reclaim any unique RawUTF8 values

**function** Count: integer;

How many items are currently stored in Value[]

**procedure** Clear;

Delete all stored RawUTF8 values

**procedure** Done;

Finalize the RawUTF8 slot - mainly its associated Safe mutex

**procedure** Init;

Initialize the RawUTF8 slot (and its Safe mutex)

**procedure** Unique(var aResult: RawUTF8; **const** aText: RawUTF8; aTextHash: cardinal);

Returns the interned RawUTF8 value

**procedure** UniqueText(var aText: RawUTF8; aTextHash: cardinal);

Ensure the supplied RawUTF8 value is interned

---

**TRawUTF8Interning** = **class**(TSynPersistent)

Allow to store only one copy of distinct RawUTF8 values
- thanks to the Copy-On-Write feature of string variables, this may reduce a lot the memory overhead of duplicated text content
- this class is thread-safe and optimized for performance

**constructor** Create(aHashTables: integer=4); **reintroduce**;

Initialize the storage and its internal hash pools
- aHashTables is the pool size, and should be a power of two <= 512
**Synopse mORMot Framework**

*Software Architecture Design 1.18*

*Date: September 16, 2020*

---

**destroyer** Destroy; **override**

*Finalize the storage*

**function** Clean(aMaxRefCount: integer=1): integer;

*Reclaim any unique RawUTF8 values*

- i.e. run a garbage collection process of all values with RefCount=1 by default, i.e. all strings which are not used any more; you may set aMaxRefCount to a higher value, depending on your expectations, i.e. 2 to delete all string which are referenced only once outside of the pool
- returns the number of unique RawUTF8 cleaned from the internal pool
- to be executed on a regular basis - but not too often, since the process can be time consuming, and void the benefit of interning

**function** Count: integer;

*How many items are currently stored in this instance*

**function** Unique(const aText: RawUTF8): RawUTF8; overload;

*Return a RawUTF8 variable stored within this class*

- if aText occurs for the first time, add it to the internal string pool
- if aText does exist in the internal string pool, return the shared instance (with its reference counter increased), to reduce memory usage

**function** Unique(aText: PUTF8Char; aTextLen: PtrInt): RawUTF8; overload;

*Return a RawUTF8 variable stored within this class from a text buffer*

- if aText occurs for the first time, add it to the internal string pool
- if aText does exist in the internal string pool, return the shared instance (with its reference counter increased), to reduce memory usage

**procedure** Clear;

*Delete any previous storage pool*

**procedure** Unique(var aResult: RawUTF8; const aText: RawUTF8); overload;

*Return a RawUTF8 variable stored within this class*

- if aText occurs for the first time, add it to the internal string pool
- if aText does exist in the internal string pool, return the shared instance (with its reference counter increased), to reduce memory usage

**procedure** Unique(var aResult: RawUTF8; aText: PUTF8Char; aTextLen: PtrInt); overload;

*Return a RawUTF8 variable stored within this class from a text buffer*

- if aText occurs for the first time, add it to the internal string pool
- if aText does exist in the internal string pool, return the shared instance (with its reference counter increased), to reduce memory usage

**procedure** UniqueText(var aText: RawUTF8);

*Ensure a RawUTF8 variable is stored within this class*

- if aText occurs for the first time, add it to the internal string pool
- if aText does exist in the internal string pool, set the shared instance (with its reference counter increased), to reduce memory usage

**procedure** UniqueVariant(var aResult: variant; const aText: RawUTF8); overload;

*Return a variant containing a RawUTF8 stored within this class*

- similar to RawUTF8ToVariant(), but with string interning
procedure UniqueVariant(var aResult: variant; aText: PUTF8Char; aTextLen: PtrInt; aAllowVarDouble: boolean=false); overload;

Return a variant, may be containing a RawUTF8 stored within this class
- similar to TextToVariant(), but with string interning
- first try with GetNumericVariantFromJSON(), then fallback to RawUTF8ToVariant() with string variable interning

procedure UniqueVariant(var aResult: variant); overload;

Ensure a variant contains only RawUTF8 stored within this class
- supplied variant should be a varString containing a RawUTF8 value

procedure UniqueVariantString(var aResult: variant; const aText: string);

Return a variant containing a RawUTF8 stored within this class
- similar to RawUTF8ToVariant(StringToUTF8()), but with string interning
- this method expects the text to be supplied as a VCL string, which will be converted into a variant containing a RawUTF8 varString instance

TSynNameValueItem = record
Store one Name/Value pair, as used by TSynNameValue class

Name: RawUTF8;
   The name of the Name/Value pair
   - this property is hashed by TSynNameValue for fast retrieval

Tag: PtrInt;
   Any associated Pointer or numerical value

Value: RawUTF8;
   The value of the Name/Value pair

TSynNameValue = object(TObject)
Pseudo-class used to store Name/Value RawUTF8 pairs
- use internaly a TDynArrayHashed instance for fast retrieval
- is therefore faster than TRawUTF8List
- is defined as an object, not as a class: you can use this in any class, without the need to destroy the content
- Delphi "object" is buggy on stack -> also defined as record with methods

Count: integer;
   The number of Name/Value pairs

DynArray: TDynArrayHashed;
   Low-level access to the internal storage hasher

List: TSynNameValueItemDynArray;
   The internal Name/Value storage

function AsCSV(const KeySeparator: RawUTF8='='; const ValueSeparator: RawUTF8='#13#10'; const IgnoreKey: RawUTF8=''): RawUTF8;

Returns all values, as CSV or INI content
function AsDocVariant(ExtendedJson: boolean=false; ValueAsString: boolean=true): variant; overload;
  Compute a TDocVariant document from the stored values

function AsJSON: RawUTF8;
  Returns all values as a JSON object of string fields

function Delete(const aName: RawUTF8): boolean;
  Search for a Name, and delete its entry in the List if it exists

function DeleteByValue(const aValue: RawUTF8; Limit: integer=1): integer;
  Search for a Value, and delete its entry in the List if it exists
  - returns the number of deleted entries
  - you may search for more than one match, by setting a >1 Limit value

function Find(const aName: RawUTF8): integer;
  Search for a Name, return the index in List
  - using fast O(1) hash algorithm

function FindByValue(const aValue: RawUTF8): integer;
  Search for a Value, return the index in List
  - using O(n) brute force algorithm with case-sensitive aValue search

function FindStart(const aUpperName: RawUTF8): integer;
  Search for the first chars of a Name, return the index in List
  - using O(n) calls of IdemPChar() function
  - here aUpperName should be already uppercase, as expected by IdemPChar()

function InitFromJSON(JSON: PUTF8Char; aCaseSensitive: boolean=false): boolean;
  Reset content, then add all fields from an JSON object
  - will first call Init() to initialize the internal array
  - then parse the incoming JSON object, storing all its field values as RawUTF8, and returning TRUE if the supplied content is correct
  - warning: the supplied JSON buffer will be decoded and modified in-place

function Initialized: boolean;
  Returns true if the Init() method has been called

function MergeDocVariant(var DocVariant: variant; ValueAsString: boolean;
  ChangedProps: PVariant=nil; ExtendedJson: boolean=false; AllowVarDouble:
  boolean=false); integer;
  Merge the stored values into a TDocVariant document
  - existing properties would be updated, then new values will be added to the supplied TDocVariant instance, ready to be serialized as a JSON object
  - if ValueAsString is TRUE, values would be stored as string
  - if ValueAsString is FALSE, numerical values would be identified by IsString() and stored as such in the resulting TDocVariant
  - if you let ChangedProps point to a TDocVariantData, it would contain an object with the stored values, just like AsDocVariant
  - returns the number of updated values in the TDocVariant, 0 if no value was changed

function Value(const aName: RawUTF8; const aDefaultValue: RawUTF8=''): RawUTF8;
  Search for a Name, return the associated Value as a UTF-8 string
function ValueBool(const aName: RawUTF8): Boolean;
   "Search for a Name, return the associated Value as boolean"
   - returns true only if the value is exactly '1'

function ValueEnum(const aName: RawUTF8; aEnumTypeInfo: pointer; out aEnum; aEnumDefault: byte=0): boolean; overload;
   "Search for a Name, return the associated Value as an enumerate"
   - returns true and set aEnum if aName was found, and associated value matched an aEnumTypeInfo item
   - returns false if no match was found

function ValueInt(const aName: RawUTF8; const aDefaultValue: Int64=0): Int64;
   "Search for a Name, return the associated Value as integer"

function ValueVariantOrNull(const aName: RawUTF8): variant;
   "Search for a Name, return the associated Value as variant"
   - returns null if the name was not found

procedure Add(const aName, aValue: RawUTF8; aTag: PtrInt=0);
   "Add an element to the array"
   - if aName already exists, its associated Value will be updated

procedure AsDocVariant(out DocVariant: variant; ExtendedJson: boolean=false; ValueAsString: boolean=true; AllowVarDouble: boolean=false); overload;
   "Compute a TDocVariant document from the stored values"
   - output variant will be reset and filled as a TDocVariant instance, ready to be serialized as a JSON object
   - if there is no value stored (i.e. Count=0), set null

procedure AsNameValues(out Names,Values: TRawUTF8DynArray);
   "Fill the supplied two arrays of RawUTF8 with the stored values"

procedure Init(aCaseSensitive: boolean);
   "Initialize the storage"
   - will also reset the internal List[] and the internal hash array

procedure InitFromCSV(CSV: PUTF8Char; NameValueSep: AnsiChar='='; ItemSep: AnsiChar=#10);
   "Reset content, then add all name=value; CSV pairs"
   - will first call Init(false) to initialize the internal array
   - if ItemSep=#10, then any kind of line feed (CRLF or LF) will be handled

procedure InitFromIniSection(Section: PUTF8Char; OnTheFlyConvert: TOnSynNameValueConvertRawUTF8=nil; OnAdd: TOnSynNameValueNotify=nil);
   "Reset content, then add all name=value pairs from a supplied .ini file section content"
   - will first call Init(false) to initialize the internal array
   - Section can be retrieved e.g. via FindSectionFirstLine()

procedure InitFromNamesValues(const Names, Values: array of RawUTF8);
   "Reset content, then add all name, value pairs"
   - will first call Init(false) to initialize the internal array

procedure SetBlobDataPtr(aValue: pointer);
   "Can be used to set all data from one BLOB memory buffer"
property BlobData: RawByteString read GetBlobData write SetBlobData;
  Can be used to set or retrieve all stored data as one BLOB content

property Bool[const aName: RawUTF8]: Boolean read GetBool;
  Search for a Name, return the associated Value as boolean
  returns true if aName stores '1' as associated value

property Int[const aName: RawUTF8]: Int64 read GetInt;
  Search for a Name, return the associated Value as integer
  returns 0 if aName is not found, or not a valid Int64 in the stored keys

property OnAfterAdd: TOnSynNameValueNotify read fOnAdd write fOnAdd;
  Event triggered after an item has just been added to the list

property Str[const aName: RawUTF8]: RawUTF8 read GetStr;
  Search for a Name, return the associated Value as a UTF-8 string
  returns '' if aName is not found in the stored keys

TLecuyer = object(TObject)
  Low-level object implementing a 32-bit Pierre L'Ecuyer software generator
  - as used by Random32gsl, and Random32 if no RDRAND hardware is available
  - is not thread-safe by itself, but cross-compiler and cross-platform, still very fast with a much
    better distribution than Delphi system's Random() function
  - Random32gsl/Random32 will use a threadvar to have thread safety

function Next(max: cardinal): cardinal; overload;
  Compute the next 32-bit generated value, in range [0..max-1]
  - will automatically reseed after around 65,000 generated values

function Next: cardinal; overload;
  Compute the next 32-bit generated value
  - will automatically reseed after around 65,000 generated values

procedure Seed(entropy: PByteArray; entropylen: PtrInt);
  Force an immediate seed of the generator from current system state
  - should be called before any call to the Next method

TJSONCustomParserRTTI = class(TObject)
  Used to store additional RTTI in TJSONCustomParser internal structures

constructor Create(const aPropertyName: RawUTF8; aPropertyType: TJSONCustomParserRTTIType);
  Initialize the instance

class function CreateFromRTTI(const PropertyName: RawUTF8; Info: pointer; ItemSize: integer): TJSONCustomParserRTTI;
  Initialize an instance from the RTTI type information
  - will return an instance of this class of any inherited class
class function CreateFromTypeName(const aPropertyName, aCustomRecordTypeName: RawUTF8): TJSONCustomParserRTTI;

Create an instance from a specified type name
- will return an instance of this class of any inherited class

function ReadOneLevel(var P: PUTF8Char; var Data: PByte; Options: TJSONCustomParserSerializationOptions; CustomVariantOptions: PDocVariantOptions): boolean;

Unserialize some JSON content into its binary internal representation
- on error, returns false and P should point to the faulty text input

class function TypeInfoToSimpleRTTIType(Info: pointer): TJSONCustomParserRTTIType;

Recognize a simple type from a supplied type information
- to be called if TypeNameToSimpleRTTIType() did fail, i.e. return ptCustom
- will return ptCustom for any complex type (e.g. a record)
- see also TypeInfoToRttiType() function

class function TypeNameToSimpleBinary(const aTypeName: RawUTF8; out aDataSize, aFieldSize: integer): boolean;

Recognize a ktBinary simple type from a supplied type name
- as registered by TTextWriter.RegisterCustomJSONSerializerFromTextBinaryType

class function TypeNameToSimpleRTTIType(const TypeName: RawUTF8): TJSONCustomParserRTTIType; overload;

Recognize a simple type from a supplied type name
- will return ptCustom for any unknown type
- see also TypeInfoToRttiType() function

class function TypeNameToSimpleRTTIType(TypeName: PShortString): TJSONCustomParserRTTIType; overload;

Recognize a simple type from a supplied type name
- will return ptCustom for any unknown type
- see also TypeInfoToRttiType() function

class function TypeNameToSimpleRTTIType(TypeName: PUTF8Char; TypeNameLen: PtrInt; ItemTypeName: PRawUTF8): TJSONCustomParserRTTIType; overload;

Recognize a simple type from a supplied type name
- will return ptCustom for any unknown type
- see also TypeInfoToRttiType() function

procedure WriteOneLevel(aWriter: TTextWriter; var P: PByte; Options: TJSONCustomParserSerializationOptions; virtual;

Serialize a binary internal representation into JSON content
- this method won't append a trailing ',' character

property CustomTypeName: RawUTF8 read fCustomTypeName;

The associated type name, e.g. for a record

property FullPropertyName: RawUTF8 read fFullPropertyName;

The property name, including all parent elements
- may be void for the Root element
- e.g. 'MainProp.SubProp'
property  NestedProperty: TJSONCustomParserRTTI read fNestedProperty;

  The nested array of properties (if any)
  - assigned only if PropertyType is [ptRecord,ptArray]
  - is either the record type of each ptArray item:
    SubProp: array of record ...
  - or one NestedProperty[0] entry with PropertyName=" and PropertyType not in
    [ptRecord,ptArray]:
    SubPropNumber: array of integer;  
    SubPropText: array of RawUTF8;

property  PropertyName: RawUTF8 read fPropertyName;
  The property name
  - may be void for the Root element
  - e.g. 'SubProp'

property  PropertyType: TJSONCustomParserRTTI read fPropertyType;
  The property type
  - support only a limited set of simple types, or ptRecord for a nested record, or ptArray for a
    nested array

TJSONCustomParserCustom = class(TJSONCustomParserRTTI)
  Used to store additional RTTI as a ptCustom kind of property

  constructor  Create(const aPropertyName, aCustomTypeName: RawUTF8); virtual;
               Initialize the instance

  function   CustomReader(P: PUTF8Char; var aValue; out EndOfObject: AnsiChar;
                      CustomVariantOptions: PDocVariantOptions): PUTF8Char; virtual; abstract;
               Abstract method to read the instance from JSON
               - should return nil on parsing error

  procedure CustomWriter(const aWriter: TTextWriter; const aValue);
          virtual; abstract;
               Abstract method to write the instance as JSON

  procedure FinalizeItem(Data: Pointer); virtual;
               Release any memory used by the instance

property  CustomTypeInfo: pointer read fCustomTypeInfo;
  The associated RTTI structure

TJSONCustomParserCustomSimple = class(TJSONCustomParserCustom)
  Used to store additional RTTI for simple type as a ptCustom kind
  - this class handle currently enumerate, TGUID or static/dynamic arrays

  constructor  Create(const aPropertyName, aCustomTypeName: RawUTF8; aCustomType: pointer);
              reintroduce;
               Initialize the instance from the given RTTI structure
constructor CreateBinary(const aPropertyName: RawUTF8; aDataSize, aFixedSize: cardinal);
    Initialize the instance for a binary blob

constructor CreateFixedArray(const aPropertyName: RawUTF8; aFixedSize: cardinal);
    Initialize the instance for a static array

destructor Destroy; override;
    Released used memory

function CustomReader(P: PUTF8Char; var aValue; out EndOfObject: AnsiChar;
    CustomVariantOptions: PDocVariantOptions): PUTF8Char; override;
    Method to read the instance from JSON

procedure CustomWriter(const aWriter: TTextWriter; const aValue); override;
    Method to write the instance as JSON

property KnownType: TJSONCustomParserCustomSimpleKnownType read fKnownType;
    Which kind of simple property this instance does refer to

property NestedArray: TJSONCustomParserRTTI read fNestedArray;
    The element type for ktStaticArray and ktDynamicArray

TJSONCustomParserCustomRecord = class(TJSONCustomParserCustom)
    Implement a reference to a registered record type
    - i.e. ptCustom kind of property, handled by the TTextWriter.RegisterCustomJSONSerializer*() internal list

constructor Create(const aPropertyName: RawUTF8; aCustomTypeIndex: integer);
    reintroduce; overload;
    Initialize the instance from the given record custom serialization index

function CustomReader(P: PUTF8Char; var aValue; out EndOfObject: AnsiChar;
    CustomVariantOptions: PDocVariantOptions): PUTF8Char; override;
    Method to read the instance from JSON

procedure CustomWriter(const aWriter: TTextWriter; const aValue); override;
    Method to write the instance as JSON

procedure FinalizeItem(Data: Pointer); override;
    Release any memory used by the instance

TJSONRecordAbstract = class(TObject)
    Used to handle additional RTTI for JSON record serialization
    - this class is used to define how a record is defined, and will work with any version of Delphi
    - this Abstract class is not to be used as-this, but contains all needed information to provide
      CustomWriter/CustumReader methods
    - you can use e.g. TJSONRecordTextDefinition for text-based RTTI manual definition, or (not yet
      provided) a version based on Delphi 2010+ new RTTI information

constructor Create;
    Initialize the class instance
**Synopse mORMot Framework**  
**Software Architecture Design 1.18**  
**Date: September 16, 2020**

```pascal
Destructor Destroy; override;

  Release used memory
  - when created via Compute() call, instances of this class are managed via a GarbageCollector()
    global list, so you do not need to free them

Function CustomReader(P: PUTF8Char; var aValue; out aValid: Boolean;
CustomVariantOptions: PDocVariantOptions): PUTF8Char;

  Callback for custom JSON unserialization
  - will follow the RTTI textual information as supplied to the constructor

Procedure CustomWriter(const aWriter: TTextWriter; const aValue);

  Callback for custom JSON serialization
  - will follow the RTTI textual information as supplied to the constructor

Property Options: TJSONCustomParserSerializationOptions read fOptions write
  fOptions;

  How this class would serialize/unserialize JSON content
  - by default, no option is defined
  - you can customize the expected options with the instance returned by
    TTextWriter.RegisterCustomJSONSerializerFromText() method, or via the
    TTextWriter.RegisterCustomJSONSerializerSetOptions() overloaded methods

Property Root: TJSONCustomParserRTTI read fRoot;

  Store the RTTI information of properties at root level
  - is one instance with PropertyType=ptRecord and PropertyName="
```

```pascal
TJSONRecordRTTI = class(TJSONRecordAbstract)

  Used to handle JSON record serialization using RTTI
  - is able to handle any kind of record since Delphi 2010, thanks to enhanced RTTI

Constructor Create(aRecordTypeInfo: pointer; aRoot: TJSONCustomParserRTTI);
  reintroduce;

  Initialize the instance
  - you should NOT use this constructor directly, but let e.g.
    TJSONCustomParsers.TryToGetFromRTTI() create it for you

Property RecordTypeInfo: pointer read fRecordTypeInfo;

  The low-level address of the enhanced RTTI
```

```pascal
TJSONRecordTextDefinition = class(TJSONRecordAbstract)

  Used to handle text-defined additional RTTI for JSON record serialization
  - is used by TTextWriter.RegisterCustomJSONSerializerFromText() method

Constructor Create(aRecordTypeInfo: pointer; const aDefinition: RawUTF8);
  reintroduce;

  Initialize a custom JSON serializer/unserializer from pseudo RTTI
  - you should NOT use this constructor directly, but call the FromCache() class function, which will
    use an internal definition cache
```

class function FromCache(aTypeInfo: pointer; const aDefinition: RawUTF8): TJSONRecordTextDefinition;

Retrieve a custom cached JSON serializer/unserializer from pseudo RTTI
- returned class instance will be cached for any further use
- the record where the data will be stored should be defined as PACKED:
  type TMyRecord = packed record
    A,B,C: integer;
    D: RawUTF8;
    E: record; // or array of record/integer/string/...
      E1,E2: double;
    end;
  end;
- only known sub types are integer, cardinal, Int64, single, double, currency, TDateTime,
  TTimeLog, RawUTF8, String, WideString, SynUnicode, or a nested record or dynamic array
- RTTI textual information shall be supplied as text, with the same format as with a pascal
  record, or with some shorter variations:
  FromCache( 'A,B,C: integer; D: RawUTF8; E: record E1,E2: double; end;' );
  FromCache( 'A,B,C: integer; D: RawUTF8; E: array of record E1,E2: double; end;' );
  'A,B,C: integer; D: RawUTF8; E: array of SynUnicode; F: array of integer'
  or a shorter alternative syntax for records and arrays:
  FromCache( 'A,B,C: integer; D: RawUTF8; E: {E1,E2: double}' );
  FromCache( 'A,B,C: integer; D: RawUTF8; E: [E1,E2: double]' );
  in fact ; could be ignored:
  FromCache( 'A,B,C:integer D:RawUTF8 E:{E1,E2:double}' );
  FromCache( 'A,B,C:integer D:RawUTF8 E:[E1,E2:double]' );
  or even : could be ignored:
  FromCache( 'A,B,C integer D RawUTF8 E{E1,E2 double}' );
  FromCache( 'A,B,C integer D RawUTF8 E[E1,E2 double]' );

property Definition: RawUTF8 read fDefinition;

The textual definition of this RTTI information

TTextWriter = class(TObject)

Simple writer to a Stream, specialized for the TEXT format
- use an internal buffer, faster than string+string
- some dedicated methods is able to encode any data with JSON/XML escape
- see TTextWriterWithEcho below for optional output redirection (for TSynLog)
- see SynTable.pas for SQL resultset export via TJSONWriter
- see mORMot.pas for proper class serialization via TJSONSerializer.WriteObject

Used for DI-2.1.2 (page 2545).

constructor Create(aStream: TStream; aBuf: pointer; aBufSize: integer); overload;

The data will be written to the specified Stream
- aStream may be nil: in this case, it MUST be set before using any Add*() method
- will use an external buffer (which may be allocated on stack)

Used for DI-2.1.2 (page 2545).
**constructor** Create(aStream: TStream; aBufSize: integer=8192); overload;

*The data will be written to the specified Stream*
- aStream may be nil: in this case, it MUST be set before using any Add(*) method
- default internal buffer size if 8192

*Used for DI-2.1.2 (page 2545).*

**constructor** CreateOwnedFileStream(const aFileName: TFileName; aBufSize: integer=8192);

*The data will be written to an external file*
- you should call explicitly FlushFinal or FlushToStream to write any pending data to the file

**constructor** CreateOwnedStream(aBufSize: integer=4096); overload;

*The data will be written to an internal TRawByteStringStream*
- TRawByteStringStream.DataString method will be used by TTextWriter.Text to retrieve directly the content without any data move nor allocation
- default internal buffer size if 4096 (enough for most JSON objects)
- consider using a stack-allocated buffer and the overloaded method

**constructor** CreateOwnedStream(var aStackBuf: TTextWriterStackBuffer; aBufSize: integer=SizeOf(TTextWriterStackBuffer)); overload;

*The data will be written to an internal TRawByteStringStream*
- will use the stack-allocated TTextWriterStackBuffer if possible
- TRawByteStringStream.DataString method will be used by TTextWriter.Text to retrieve directly the content without any data move nor allocation

**constructor** CreateOwnedStream(aBuf: pointer; aBufSize: integer); overload;

*The data will be written to an internal TRawByteStringStream*
- will use an external buffer (which may be allocated on stack)
- TRawByteStringStream.DataString method will be used by TTextWriter.Text to retrieve directly the content without any data move nor allocation

**destructor** Destroy; override;

*Release all internal structures*
- e.g. free fStream if the instance was owned by this class

**function** AddJSONReformat(JSON: PUTF8Char; Format: TTextWriterJSONFormat; EndOfObject: PUTF8Char): PUTF8Char;

*Append a JSON value, array or document, in a specified format*
- will parse the JSON buffer and write its content with proper line feeds and indentation, according to the supplied TTextWriterJSONFormat
- see also JSONReformat() and JSONBufferReformat() wrappers
- this method is called recursively to handle all kind of JSON values
- WARNING: the JSON buffer is decoded in-place, so will be changed
- returns the end of the current JSON converted level, or nil if the supplied content was not valid JSON
function AddJSONToXML(JSON: PUTF8Char; ArrayName: PUTF8Char=nil; EndOfObject: PUTF8Char=nil): PUTF8Char;

Append a JSON value, array or document as simple XML content
- you can use JSONBufferToXML() and JSONToXML() functions as wrappers
- this method is called recursively to handle all kind of JSON values
- WARNING: the JSON buffer is decoded in-place, so will be changed
- returns the end of the current JSON converted level, or nil if the supplied content was not correct JSON

class function GetCustomJSONParser(var DynArray: TDynArray; out CustomReader: TDynArrayJSONCustomReader; out CustomWriter: TDynArrayJSONCustomWriter): boolean;

Retrieve low-level custom serialization callbacks for a dynamic array
- returns TRUE if this array has a custom JSON parser, and set the corresponding serialization/unserialization callbacks

function InternalJSONWriter: TTextWriter;

Gives access to an internal temporary TTextWriter
- may be used to escape some JSON escaped value (i.e. escape it twice), in conjunction with AddJSONEscape(Source: TTextWriter)

function LastChar: AnsiChar;

Return the last char appended
- returns #0 if no char has been written yet

function PendingBytes: PtrUInt;

How many bytes are currently in the internal buffer and not on disk
- see TextLength for the total number of bytes, on both disk and memory

class function RegisterCustomJSONSerializerFindParser( aTypeInfo: pointer; aAddIfNotExisting: boolean=false): TJSONRecordAbstract;

Retrieve a previously registered custom parser instance from its type
- will return nil if the type info was not available, or defined just with some callbacks
- if AddIfNotExisting is TRUE, and enhanced RTTI is available (since Delphi 2010), you would be able to retrieve this type's parser even if the record type has not been previously used
class function RegisterCustomJSONSerializerFromText(aTypeInfo: pointer; const aRTTIDefinition: RawUTF8): TJSONRecordAbstract; overload;

*Define a custom serialization for a given dynamic array or record*
- the RTTI information will here be defined as plain text
- since Delphi 2010, you can call directly RegisterCustomJSONSerializerFromTextSimpleType()
- a TypeInfo may be valid TypeInfo(), or any fixed pointer value if the record does not have any RTTI (e.g. a record without any nested reference-counted types)
- the record where the data will be stored should be defined as PACKED:

```pascal
type TMyRecord = packed record
  A, B, C: integer;
  D: RawUTF8;
  E: record; // or array of record/integer/string/...
    E1, E2: double;
  end;
end;
```
- call this method with aRTTIDefinition='"' to return back to the default binary + Base64 encoding serialization (i.e. undefined custom serializer)
- only known sub types are byte, word, integer, cardinal, Int64, single, double, currency, TDateTime, TTimeLog, RawUTF8, String, WideString, SynUnicode, TGUID (encoded via GUIDToText) or a nested record or dynamic array of the same simple types or record
- RTTI textual information shall be supplied as text, with the same format as with a Pascal record:

```pascal
'A,B,C: integer; D: RawUTF8; E: record E1, E2: double;'
'A,B,C: integer; D: RawUTF8; E: array of record E1, E2: double;'
'A,B,C: integer; D: RawUTF8; E: array of SynUnicode; F: array of TGUID'
```

or a shorter alternative syntax for records and arrays:

```pascal
'A,B,C: integer; D: RawUTF8; E: {E1, E2: double}'
'A,B,C: integer; D: RawUTF8; E: [E1, E2: double]'
```

in fact, could be ignored:

```pascal
'A,B,C:integer D:RawUTF8 E::{E1, E2:double}'
'A,B,C:integer D:RawUTF8 E::{E1, E2:double}'
```
or even, could be ignored:

```pascal
'A,B,C integer D:RawUTF8 E{E1, E2 double}'
'A,B,C integer D:RawUTF8 E{E1, E2 double}'
```
- it will return the cached TJSONRecordTextDefinition instance corresponding to the supplied RTTI text definition

class function RegisterCustomJSONSerializerSetOptions(const aTypeInfo: array of pointer; aOptions: TJSONCustomParserSerializationOptions; aAddIfNotExisting: boolean=false): boolean; overload;

*Change options for custom serialization of dynamic arrays or records*
- will return TRUE if the options have been changed, FALSE if the supplied type info was not previously registered for at least one type
- if AddIfNotExisting is TRUE, and enhanced RTTI is available (since Delphi 2010), you would be able to customize the options of this type
### class function RegisterCustomJSONSerializerSetOptions

```pascal
function RegisterCustomJSONSerializerSetOptions(aTypeInfo: pointer; aOptions: TJSONCustomParserSerializationOptions; aAddIfNotExisting: boolean=false): boolean; overload;
```

*Change options for custom serialization of dynamic array or record*
- will return TRUE if the options have been changed, FALSE if the supplied type info was not previously registered
- if AddIfNotExisting is TRUE, and enhanced RTTI is available (since Delphi 2010), you would be able to customize the options of this type

### function Text: RawUTF8;

*Retrieve the data as a string*

### procedure Add(const Format: RawUTF8; const Values: array of const; Escape: TTextWriterKind=twNone; WriteObjectOptions: TTextWriterWriteObjectOptions=[woFullExpand]); overload;

*Append strings or integers with a specified format*
- `%` = #37 marks a string, integer, floating-point, or class parameter to be appended as text (e.g. class name)
- if StringEscape is false (by default), the text won't be escaped before adding; but if set to true text will be JSON escaped at writing
- note that due to a limitation of the "array of const" format, cardinal values should be type-casted to Int64() - otherwise the integer mapped value will be transmitted, therefore wrongly

### procedure Add(Value: Extended; precision: integer; noexp: boolean=false); overload;

*Append a floating-point Value as a String*
- write "Infinity", "-Infinity", and "NaN" for corresponding IEEE values
- noexp=true will call ExtendedToShortNoExp() to avoid any scientific notation in the resulting text

### procedure Add(const guid: TGUID); overload;

*Append a GUID value, encoded as text without any {}*
- will store e.g. '3F2504E0-4F89-11D3-9A0C-0305E82C3301'

### procedure Add(const Values: array of const); overload;

*Append some values at once*
- text values (e.g. RawUTF8) will be escaped as JSON

### procedure Add(const V: TVarRec; Escape: TTextWriterKind=twNone; WriteObjectOptions: TTextWriterWriteObjectOptions=[woFullExpand]); overload;

*Append an open array constant value to the buffer*
- "" won't be added for string values
- string values may be escaped, depending on the supplied parameter
- very fast (avoid most temporary storage)

### procedure Add(P: PUTF8Char; Len: PtrInt; Escape: TTextWriterKind); overload;

*Write some #0 ended UTF-8 text, according to the specified format*
- if Escape is a constant, consider calling directly AddNoJSONEscape, AddJSONEscape or AddOnSameLine methods
procedure Add(P: PUTF8Char; Escape: TTextWriterKind); overload;
   Write some #0 ended UTF-8 text, according to the specified format
   - if Escape is a constant, consider calling directly AddNoJSONEscape, AddJSONEscape or AddOnSameLine methods

procedure Add(c1,c2: AnsiChar); overload;
   Append two chars to the buffer

procedure Add(Value: Int64); overload;
   Already implemented by Add(Value: PPtrInt) method append a 64-bit signed Integer Value as text

procedure Add(Value: PPtrInt); overload;
   Append a 32-bit signed Integer Value as text

procedure Add(c: AnsiChar); overload;
   Append one ASCII char to the buffer

procedure Add(Value: boolean); overload;
   Append a boolean Value as text
   - write either 'true' or 'false'

procedure Add2(Value: PPtrUInt);
   Append an Integer Value as a 2 digits String with comma

procedure Add3(Value: PPtrUInt);
   Append an Integer Value as a 3 digits String without any added comma

procedure Add4(Value: PPtrUInt);
   Append an Integer Value as a 4 digits String with comma

procedure AddAnsiString(const s: AnsiString; Escape: TTextWriterKind); overload;
   Append some UTF-8 encoded chars to the buffer, from the main AnsiString type
   - use the current system code page for AnsiString parameter

procedure AddAnyAnsiBuffer(P: PAnsiChar; Len: PPtrInt; Escape: TTextWriterKind;
   CodePage: Integer);
   Append some UTF-8 encoded chars to the buffer, from any Ansi buffer
   - the codepage should be specified, e.g. CP_UTF8, CP_RAWBYTESTRING, CODEPAGE_US, or any version supported by the Operating System
   - if codepage is 0, the current CurrentAnsiConvert.CodePage would be used
   - will use TSynAnsiConvert to perform the conversion to UTF-8

procedure AddAnyAnsiString(const s: RawByteString; Escape: TTextWriterKind;
   CodePage: Integer=-1);
   Append some UTF-8 encoded chars to the buffer, from any AnsiString value
   - if CodePage is left to its default value of -1, it will assume CurrentAnsiConvert.CodePage prior to Delphi 2009, but newer UNICODE versions of Delphi will retrieve the code page from string
   - if CodePage is defined to a >= 0 value, the encoding will take place

procedure AddBinToHex(Bin: Pointer; BinBytes: integer);
   Append some binary data as hexadecimal text conversion
procedure AddBinToHexDisplay(Bin: pointer; BinBytes: integer);
   Fast conversion from binary data into hexa chars, ready to be displayed
   - using this function with Bin^ as an integer value will serialize it in big-endian order
     (most-significant byte first), as used by humans
   - up to the internal buffer bytes may be converted

procedure AddBinToHexDisplayLower(Bin: pointer; BinBytes: integer);
   Fast conversion from binary data into MSB hexa chars
   - up to the internal buffer bytes may be converted

procedure AddBinToHexDisplayMinChars(Bin: pointer; BinBytes: PtrInt);
   Append a Value as significant hexadecimal text
   - append its minimal size, i.e. excluding highest bytes containing 0
   - use GetNextItemHexa() to decode such a text value

procedure AddBinToHexDisplayQuoted(Bin: pointer; BinBytes: integer);
   Fast conversion from binary data into quoted MSB lowercase hexa chars
   - up to the internal buffer bytes may be converted

procedure AddByteToHex(Value: byte);
   Write a byte as hexa chars

procedure AddChars(aChar: AnsiChar; aCount: integer);
   Write the same character multiple times

procedure AddClassName(aClass: TClass);
   Append the class name of an Object instance as text
   - aClass must be not nil

procedure AddCR;
   Append CR+LF (#13#10) chars
   - this method won't call EchoAdd() registered events - use AddEndOfLine() method instead
   - AddEndOfLine() will append either CR+LF (#13#10) or LF (#10) depending on a flag

procedure AddCRAndIndent;
   Append CR+LF (#13#10) chars and #9 indentation
   - indentation depth is defined by fHumanReadableLevel protected field

procedure AddCSVConst(const Values: array of const);
   Append an array of const as CSV of JSON values

procedure AddCSVDouble(const Doubles: array of double); overload;
   Append an array of doubles as CSV

procedure AddCSVInteger(const Integers: array of Integer); overload;
   Append an array of integers as CSV

procedure AddCSVUTF8(const Values: array of RawUTF8); overload;
   Append an array of RawUTF8 as CSV of JSON strings

procedure AddCurr64(const Value: currency); overload;
   Append a Currency from its Int64 in-memory representation

procedure AddCurr64(const Value: Int64); overload;
   Append a Currency from its Int64 in-memory representation
procedure AddCurrentLogTime(LocalTime: boolean);
  *Append the current UTC date and time, in our log-friendly format*
  - e.g. append '20110325 19241502' - with no trailing space nor tab
  - you may set LocalTime=TRUE to write the local date and time instead
  - this method is very fast, and avoid most calculation or API calls

procedure AddCurrentNCSALogTime(LocalTime: boolean);
  *Append the current UTC date and time, in our log-friendly format*
  - e.g. append '19/Feb/2019:06:18:55 ' - including a trailing space
  - you may set LocalTime=TRUE to write the local date and time instead
  - this method is very fast, and avoid most calculation or API calls

procedure AddDateTime(Value: PDateTime; FirstChar: AnsiChar='T'; QuoteChar: AnsiChar=#0; WithMS: boolean=false); overload;
  *Append a TDateTime value, expanded as Iso-8601 encoded text*
  - use 'YYYY-MM-DDThh:mm:ss' format (with FirstChar='T')
  - if WithMS is TRUE, will append '.sss' for milliseconds resolution
  - if QuoteChar is not #0, it will be written before and after the date

procedure AddDateTime(const Value: TDateTime; WithMS: boolean=false); overload;
  *Append a TDateTime value, expanded as Iso-8601 encoded text*
  - use 'YYYY-MM-DDThh:mm:ss' format
  - append nothing if Value=0
  - if WithMS is TRUE, will append '.sss' for milliseconds resolution

procedure AddDateTimeMS(const Value: TDateTime; Expanded: boolean=true; FirstTimeChar: AnsiChar = 'T'; const TZD: RawUTF8 = 'Z');
  *Append a TDateTime value, expanded as Iso-8601 text with milliseconds and Time Zone designator*
  - i.e. 'YYYY-MM-DDThh:mm:ss.sssZ' format
  - TZD is the ending time zone designator ('', 'Z' or '+hh:mm' or '-hh:mm')

procedure AddDouble(Value: double; noexp: boolean=false);
  *Append a floating-point Value as a String*
  - write "Infinity", "-Infinity", and "NaN" for corresponding IEEE values
  - noexp=true will call ExtendedToShortNoExp() to avoid any scientific notation in the resulting text
procedure AddDynArrayJSON(var aDynArray: TDynArray); overload;

Append a dynamic array content as UTF-8 encoded JSON array
- expect a dynamic array TDynArray wrapper as incoming parameter
- TIntDynArray, TInt64DynArray, TCardinalDynArray, TDoubleDynArray, TCurrencyDynArray,
  TWordDynArray and TByteDynArray will be written as numerical JSON values
- TRawUTF8DynArray, TWinAnsiDynArray, TRawByteDynArray, TStringDynArray,
  TStringDynArray, TSynUnicodeDynArray, TTimeLogDynArray, and TDateTimeDynArray will
  be written as escaped UTF-8 JSON strings (and Iso-8601 textual encoding if necessary)
- you can add some custom serializers via RegisterCustomJSONSerializer() class method, to
  serialize any dynamic array as valid JSON
- any other non-standard or non-registered kind of dynamic array (including array of records)
  will be written as Base64 encoded binary stream, with a JSON_BASE64_MAGIC prefix (UTF-8
  encoded \uFFF0 special code) - this will include TBytes (i.e. array of bytes) content, which is a
  good candidate for BLOB stream
- typical content could be
  "[1,2,3,4]" or "[\uFFF0base64encodedbinary]"
- by default, custom serializers defined via RegisterCustomJSONSerializer() would write
  enumerates and sets as integer numbers, unless twoEnumSetsAsTextInRecord is set in the
  instance Options

procedure AddDynArrayJSON(var aDynArray: TDynArrayHashed); overload;

Append a dynamic array content as UTF-8 encoded JSON array
- expect a dynamic array TDynArrayHashed wrapper as incoming parameter

procedure AddDynArrayJSON(aTypeInfo: pointer; const aValue); overload;

Append a dynamic array content as UTF-8 encoded JSON array
- just a wrapper around the other overloaded method, creating a temporary TDynArray wrapper
  on the stack
- to be used e.g. for custom record JSON serialization, within a TDynArrayJSONCustomWriter
  callback

procedure AddDynArrayJSONAsString(aTypeInfo: pointer; var aValue);

Same as AddDynArrayJSON(), but will double all internal " and bound with "
- this implementation will avoid most memory allocations

procedure AddFieldName(const FieldName: RawUTF8);

Append a RawUTF8 property name, as ""FieldName"":'
- FieldName content should not need to be JSON escaped (e.g. no " within)
- if twoForceJSONExtended is defined in CustomOptions, it would append 'PropName:' without
  the double quotes
- is a wrapper around AddProp()

procedure AddFloatStr(P: PUTF8Char);

Append a floating-point text buffer
- will correct on the fly '.5' -> '0.5' and ' - .5' -> '-0.5'
- will end not only on #0 but on any char not matching 1[.2[e[-3]]] pattern
- is used when the input comes from a third-party source with no regular output, e.g. a database
  driver

procedure AddHtmlEscape(Text: PUTF8Char; TextLen: PtrInt; Fmt:
TTextWriterHTMLFormat=hfAnyWhere); overload;

Append some chars, escaping all HTML special chars as expected
Synopse mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

procedure AddHtmlEscape(Text: PUTF8Char; Fmt: TTextWriterHTMLFormat=hfAnyWhere);
overload;

Append some chars, escaping all HTML special chars as expected
procedure AddHtmlEscapeString(const Text: string; Fmt:
TTextWriterHTMLFormat=hfAnyWhere);

Append some chars, escaping all HTML special chars as expected
procedure AddHtmlEscapeUTF8(const Text: RawUTF8; Fmt:
TTextWriterHTMLFormat=hfAnyWhere);

Append some chars, escaping all HTML special chars as expected
procedure AddInstanceName(Instance: TObject; SepChar: AnsiChar);

Append an Instance name and pointer, as '"TObjectList(00425E68)"'+SepChar
- Instance must be not nil
procedure AddInstancePointer(Instance: TObject; SepChar: AnsiChar;
IncludeUnitName, IncludePointer: boolean); virtual;

Append an Instance name and pointer, as 'TObjectList(00425E68)'+SepChar
- Instance must be not nil
- overriden version in TJSONSerializer would implement IncludeUnitName
procedure AddInt18ToChars3(Value: cardinal);

Write a Int18 value (0..262143) as 3 chars
- this encoding is faster than Base64, and has spaces on the left side
- use function Chars3ToInt18() to decode the textual content
procedure AddJSON(const Format: RawUTF8; const Args,Params: array of const);

Encode the supplied (extended) JSON content, with parameters, as an UTF-8 valid JSON object
content
- in addition to the JSON RFC specification strict mode, this method will handle some BSON-like
extensions, e.g. unquoted field names:
aWriter.AddJSON('{id:?,%:{name:?,birthyear:?}}',['doc'],[10,'John',1982]);

- you can use nested _Obj() / _Arr() instances
aWriter.AddJSON('{%:{$in:[?,?]}}',['type'],['food','snack']);
aWriter.AddJSON('{type:{$in:?}}',[],[_Arr(['food','snack'])]);
// which are the same as:
aWriter.AddShort('{"type":{"$in":["food","snack"]}}');

- if the SynMongoDB unit is used in the application, the MongoDB Shell syntax will also be
recognized to create TBSONVariant, like
new Date()

ObjectId()

MinKey

MaxKey

/<jRegex>/<jOptions>

see @http://docs.mongodb.org/manual/reference/mongodb-extended-json
aWriter.AddJSON('{name:?,field:/%/i}',['acme.*corp'],['John']))
// will write
'{"name":"John","field":{"$regex":"acme.*corp","$options":"i"}}'

- will call internally _JSONFastFmt() to create a temporary TDocVariant with all its features - so is
slightly slower than other AddJSON* methods

SynCommons.pas unit - Rev. 1.18

Page 779 of 2552


procedure AddJSONArraysAsJSONObject(keys,values: PUTF8Char);

Append two JSON arrays of keys and values as one JSON object
- i.e. makes the following transformation:
  [
    [key1,key2,...] + [value1,value2,...]
  ] -> {key1:value1,key2,value2,...}
- this method won't allocate any memory during its process, nor modify the keys and values
  input buffers
- is the reverse of the JSONObjectAsJSONArray() function

procedure AddJSONEscape(const V: TVarRec); overload;

Append an open array constant value to the buffer
- "" will be added if necessary
- escapes chars according to the JSON RFC
- very fast (avoid most temporary storage)

Used for DI-2.1.2 (page 2545).

procedure AddJSONEscape(P: Pointer; Len: PtrInt=0); overload;

Append some UTF-8 encoded chars to the buffer
- escapes chars according to the JSON RFC
- if Len is 0, writing will stop at #0 (default Len=0 is slightly faster than specifying Len>0 if you are
  sure P is zero-ended - e.g. from RawUTF8)

Used for DI-2.1.2 (page 2545).

procedure AddJSONEscape(const NameValuePairs: array of const); overload;

Encode the supplied data as an UTF-8 valid JSON object content
- data must be supplied two by two, as Name,Value pairs, e.g.
  aWriter.AddJSONEscape(['name','John','year',1972]);
- will append to the buffer:
  '{"name":"John","year":1972}'
- or you can specify nested arrays or objects with '{...}' or '{...}':
  aWriter.AddJSONEscape(['doc',
    '{','"name":"John","ab":["a","b"],"id":123}',
    'id',123]);
- will append to the buffer:
  '{"doc":{"name":"John","abc":"["a","b"]","id":123}}'
- note that, due to a Delphi compiler limitation, cardinal values should be type-casted to Int64()
  (otherwise the integer mapped value will be converted)
- you can pass nil as parameter for a null JSON value

Used for DI-2.1.2 (page 2545).

procedure AddJSONEscape(Source: TTextWriter); overload;

Flush a supplied TTextWriter, and write pending data as JSON escaped text
- may be used with InternalJSONWriter, as a faster alternative to
  AddJSONEscape(Pointer(fInternalJSONWriter.Text),0);

Used for DI-2.1.2 (page 2545).

procedure AddJSONEscapeAnsiString(const s: AnsiString);

Append some UTF-8 encoded chars to the buffer, from the main AnsiString type
- escapes chars according to the JSON RFC
procedure AddJSONEscapeString(const s: string);

*Append some UTF-8 encoded chars to the buffer, from a generic string type*
- faster than AddJSONEscape(pointer(StringToUTF8(string))
- escapes chars according to the JSON RFC

procedure AddJSONEscapeW(P: PWord; Len: PtrInt=0);

*Append some Unicode encoded chars to the buffer*
- if Len is 0, Len is calculated from zero-ended widechar
- escapes chars according to the JSON RFC

procedure AddJSONString(const Text: RawUTF8);

*Append a UTF-8 JSON String, between double quotes and with JSON escaping*

procedure AddLine(const Text: shortstring);

*Append a line of text with CR+LF at the end*

procedure AddMicroSec(MS: cardinal);

*Append a time period, specified in micro seconds, in 00.000.000 TSynLog format*

procedure AddNoJSONEscape(P: Pointer; Len: PtrInt);

*Append some UTF-8 chars to the buffer*
- don't escapes chars according to the JSON RFC

procedure AddNoJSONEscape(Source: TTextWriter); overload;

*Flush a supplied TTextWriter, and write pending data as JSON escaped text*
- may be used with InternalJSONWriter, as a faster alternative to AddNoJSONEscapeUTF8(Source.Text);

procedure AddNoJSONEscape(P: Pointer); overload;

*Append some UTF-8 chars to the buffer*
- input length is calculated from zero-ended char
- don't escapes chars according to the JSON RFC

procedure AddNoJSONEscapeString(const s: string);

*Append some UTF-8 encoded chars to the buffer, from a generic string type*
- faster than AddNoJSONEscape(pointer(StringToUTF8(string))
- don't escapes chars according to the JSON RFC
- will convert the Unicode chars into UTF-8

procedure AddNoJSONEscapeUTF8(const text: RawByteString);

*Append some UTF-8 chars to the buffer*
- don't escapes chars according to the JSON RFC

procedure AddNoJSONEscapeW(WideChar: PWord; WideCharCount: integer);

*Append some unicode chars to the buffer*
- WideCharCount is the unicode chars count, not the byte size
- don't escapes chars according to the JSON RFC
- will convert the Unicode chars into UTF-8

procedure AddObjArrayJSON(const aObjArray; aOptions: TTextWriterWriteObjectOptions=[woDontStoreDefault]);

*Append a T*^ObjArray dynamic array as a JSON array*
- as expected by TJSONSerializer.RegisterObjArrayForJSON()
procedure AddOnce(c: AnsiChar); overload;
  Append one ASCII char to the buffer, if not already there as LastChar

procedure AddOnSameLine(P: PUTF8Char; Len: PtrInt); overload;
  Append some chars to the buffer in one line
  - will write #0..#31 chars as spaces (so content will stay on the same line)

procedure AddOnSameLine(P: PUTF8Char); overload;
  Append some chars to the buffer in one line
  - P should be ended with a #0
  - will write #1..#31 chars as spaces (so content will stay on the same line)

procedure AddOnSameLineW(P: PWord; Len: PtrInt);
  Append some wide chars to the buffer in one line
  - will write #0..#31 chars as spaces (so content will stay on the same line)

procedure AddPointer(P: PtrUInt);
  Add the pointer into significant hexa chars, ready to be displayed

procedure AddProp(PropName: PUTF8Char; PropNameLen: PtrInt);
  Append a property name, as ""PropName":'
  - PropName content should not need to be JSON escaped (e.g. no " within, and only ASCII 7-bit characters)
  - if twoForceJSONExtended is defined in CustomOptions, it would append 'PropName:' without the double quotes

procedure AddPropJSONInt64(const PropName: shortstring; Value: Int64);
  Append a JSON field name, followed by a number value and a comma (',')

procedure AddPropJSONString(const PropName: shortstring; const Text: RawUTF8);
  Append a JSON field name, followed by an escaped UTF-8 JSON String and a comma (',')

procedure AddPropName(const PropName: ShortString);
  Append a ShortString property name, as ""PropName":'
  - PropName content should not need to be JSON escaped (e.g. no " within, and only ASCII 7-bit characters)
  - if twoForceJSONExtended is defined in CustomOptions, it would append 'PropName:' without the double quotes
  - is a wrapper around AddProp()

procedure AddQ(Value: QWord);
  Append an Unsigned 64-bit Integer Value as a String

procedure AddQHex(Value: Qword);
  Append an Unsigned 64-bit Integer Value as a quoted hexadecimal String

procedure AddQuotedStr(Text: PUTF8Char; Quote: AnsiChar; TextMaxLen: PtrInt=0);
  Append some UTF-8 text, quoting all " chars
  - same algorithm than AddString(QuotedStr()) - without memory allocation, and with an optional maximum text length (truncated with ending '...')
  - this function implements what is specified in the official SQLite3 documentation: "A string constant is formed by enclosing the string in single quotes ('). A single quote within the string can be encoded by putting two single quotes in a row - as in Pascal."
procedure AddQuotedStringAsJSON(const QuotedString: RawUTF8);
  Append a quoted string as JSON, with in-place decoding
  - if QuotedString does not start with ' or ", it will written directly (i.e. expects to be a number, or
    null/true/false constants)
  - as used e.g. by TJSONObjectDecoder.EncodeAsJSON method and JSONEncodeNameSQLValue() function

procedure AddRawJSON(const json: RawJSON);
  Append some UTF-8 chars to the buffer
  - if supplied json is ", will write 'null'

procedure AddRecordJSON(const Rec; TypeInfo: pointer);
  Append a record content as UTF-8 encoded JSON or custom serialization
  - default serialization will use Base64 encoded binary stream, or a custom serialization, in case of
    a previous registration via RegisterCustomJSONSerializer() class method - from a dynamic array
    handling this kind of records, or directly from TypeInfo() of the record
  - by default, custom serializers defined via RegisterCustomJSONSerializer() would write
    enumerates and sets as integer numbers, unless twoEnumSetsAsTextInRecord or
    twoEnumSetsAsBooleanInRecord is set in the instance CustomOptions

procedure AddReplace(Text: PUTF8Char; Orig, Replaced: AnsiChar);
  Append some chars, replacing a given character with another

procedure AddShort(const Text: ShortString);
  Append a ShortString

procedure AddSingle(Value: single; noexp: boolean=false);
  Append a floating-point Value as a String
  - write "Infinity", "-Infinity", and "NaN" for corresponding IEEE values
  - noexp=true will call ExtendedToShortNoExp() to avoid any scientific notation in the resulting text

procedure AddString(const Text: RawUTF8);
  Append an UTF-8 String, with no JSON escaping

procedure AddStringCopy(const Text: RawUTF8; start, len: PtrInt);
  Append a sub-part of an UTF-8 String
  - emulates AddString(copy(Text, start, len))

procedure AddStrings(const Text: array of RawUTF8); overload;
  Append several UTF-8 strings

procedure AddStrings(const Text: RawUTF8; count: integer); overload;
  Append an UTF-8 string several times

procedure AddTimeLog(Value: PInt64);
  Append a TTimeLog value, expanded as Iso-8601 encoded text

procedure AddTrimLeftLowerCase(Text: PShortString);
  Append after trim first lowercase chars ("otDone" will add 'Done' e.g.)

procedure AddTrimSpaces(P: PUTF8Char); overload;
  Append a UTF-8 String excluding any space or control char
  - this won't escape the text as expected by JSON
procedure AddTrimSpaces(const Text: RawUTF8); overload;
- Append a UTF-8 String excluding any space or control char
- this won't escape the text as expected by JSON

procedure AddTypedJSON(aTypeInfo: pointer; const aValue);
- Append a JSON value from its RTTI type
- handle tkClass,tkEnumeration,tkSet,tkRecord,tkDynArray,tkVariant types
- write null for other types

procedure AddU(Value: cardinal);
- Append an Unsigned 32-bit Integer Value as a String

procedure AddUnixMSTime(Value: PInt64; WithMS: boolean=false);
- Append a TUnixMSTime value, expanded as Iso-8601 encoded text

procedure AddUnixTime(Value: PInt64);
- Append a TUnixTime value, expanded as Iso-8601 encoded text

procedure AddVariant(const Value: variant; Escape: TTextWriterKind=twJSONEscape);
- Append a variant content as number or string
  - default Escape=twJSONEscape will create valid JSON content, which can be converted back to a
    variant value using VariantLoadJSON()
  - default JSON serialization options would apply, unless twoForceJSONExtended or
twoForceJSONStandard is defined
  - note that before Delphi 2009, any varString value is expected to be a RawUTF8 instance - which
does make sense in the mORMot context

procedure AddVoidRecordJSON(TypeInfo: pointer);
- Append a void record content as UTF-8 encoded JSON or custom serialization
  - this method will first create a void record (i.e. filled with #0 bytes) then save its content with
    default or custom serialization

procedure AddW(P: PWord; Len: PInt; Escape: TTextWriterKind);
- Write some #0 ended Unicode text as UTF-8, according to the specified format
  - if Escape is a constant, consider calling directly AddNoJSONEscapeW, AddJSONEscapeW or
    AddOnSameLineW methods

procedure AddXmlEscape(Text: PUTF8Char);
- Append some chars, escaping all XML special chars as expected
  - i.e. < > & '' as &lt; &gt; &amp; &quote; &apos;
  - and all control chars (i.e. #1..#31) as &#..;
  - see @http://www.w3.org/TR/xml/#syntax

procedure CancelAll;
- Rewind the Stream to the position when Create() was called
  - note that this does not clear the Stream content itself, just move back its writing position to its
    initial place

procedure CancelLastChar(aCharToCancel: AnsiChar); overload;
- The last char appended is canceled, if match the supplied one
  - only one char cancellation is allowed at the same position: don't call
    CancelLastChar/CancelLastComma more than once without appending text inbetween
procedure CancelLastChar; overload;
    The last char appended is canceled
    - only one char cancellation is allowed at the same position: don't call
      CancelLastChar/CancelLastComma more than once without appending text inbetween

procedure CancelLastComma;
    The last char appended is canceled if it was a ','
    - only one char cancellation is allowed at the same position: don't call
      CancelLastChar/CancelLastComma more than once without appending text inbetween

procedure FlushFinal;
    Write pending data to the Stream, without automatic buffer resize
    - will append the internal memory buffer to the Stream
    - in short, FlushToStream may be called during the adding process, and FlushFinal at the end of
      the process, just before using the resulting Stream
    - if you don't call FlushToStream or FlushFinal, some pending characters may not be copied to
      the Stream: you should call it before using the Stream

procedure FlushToStream; virtual;
    Write pending data to the Stream, with automatic buffer resize
    - you should not have to call FlushToStream in most cases, but FlushFinal at the end of the
      process, just before using the resulting Stream
    - FlushToStream may be used to force immediate writing of the internal memory buffer to the
      destination Stream
    - you can set FlushToStreamNoAutoResize=true or call FlushFinal if you do not want the
      automatic memory buffer resize to take place

procedure ForceContent(const text: RawUTF8);
    Set the internal stream content with the supplied UTF-8 text

class procedure RegisterCustomJSONSerializer(aTypeInfo: pointer; aReader: TDynArrayJSONCustomReader; aWriter: TDynArrayJSONCustomWriter);
    Define a custom serialization for a given dynamic array or record
    - expects TypeInfo() from a dynamic array or a record (will raise an exception otherwise)
    - for a dynamic array, the associated item record RTTI will be registered
    - for a record, any matching dynamic array will also be registered
    - by default, TIntegerDynArray and such known classes are processed as true JSON arrays: but
      you can specify here some callbacks to perform the serialization process for any kind of dynamic
      array
    - any previous registration is overridden
    - setting both aReader=aWriter=nil will return back to the default binary + Base64 encoding
      serialization (i.e. undefine custom serializer)

class procedure RegisterCustomJSONSerializerForVariant(aClass: TCustomVariantType; aReader: TDynArrayJSONCustomReader; aWriter: TDynArrayJSONCustomWriter);
    Define a custom serialization for a given variant custom type
    - used e.g. to serialize TBCD values

class procedure RegisterCustomJSONSerializerForVariantByType(aVarType: TVarType; aReader: TDynArrayJSONCustomReader; aWriter: TDynArrayJSONCustomWriter);
    Define a custom serialization for a given variant custom type
    - used e.g. to serialize TBCD values
Define a custom serialization for several dynamic arrays or records
- the TypeInfo() and textual RTTI information will here be defined as
  ([TypeInfo(TType1),_TType1,TypeInfo(TType2),_TType2]) pairs
- a wrapper around the overloaded RegisterCustomJSONSerializerFromText()

Define a custom binary serialization for a given simple type
- you should be able to use this type in the RTTI text definition of any further
  RegisterCustomJSONSerializerFromText() call
- data will be serialized as BinToHexDisplayLower() JSON hexadecimal string
- you can truncate the original data size (e.g. if all bits of an integer are not used) by specifying
  the aFieldSize optional parameter

Define custom binary serialization for several simple types
- data will be serialized as BinToHexDisplayLower() JSON hexadecimal string
- the TypeInfo() and associated size information will here be defined as triplets:
  ([TypeInfo(TType1),SizeOf(TType1),TYPE1_BYTES,TypeInfo(TType2),SizeOf(TType2),TYPE2getBytes ])
- a wrapper around the overloaded RegisterCustomJSONSerializerFromTextBinaryType()

Define a custom serialization for a given simple type
- you should be able to use this type in the RTTI text definition of any further
  RegisterCustomJSONSerializerFromText() call
- the RTTI information should be enough to serialize the type from its name (e.g. an
  enumeration for older Delphi revision, but all records since Delphi 2010)
- you can supply a custom type name, which will be registered in addition to the "official" name
  defined at RTTI level
- on older Delphi versions (up to Delphi 2009), it will handle only enumerations, which will be
  transmitted as JSON string instead of numbers
- since Delphi 2010, any record type can be supplied - which is more convenient than calling
  RegisterCustomJSONSerializerFromText()

Define a custom serialization for several simple types
- will call the overloaded RegisterCustomJSONSerializerFromTextSimpleType method for each
  supplied type information
class procedure SetDefaultEnumTrim(aShouldTrimEnumsAsText: boolean);

  Allow to override the default JSON serialization of enumerations and sets as text, which would write the whole identifier (e.g. 'sllError')
  - calling SetDefaultEnumTrim(true) would force the enumerations to be trimmed for any lower case char, e.g. sllError -> 'Error'
  - this is global to the current process, and should be use mainly for compatibility purposes for the whole process
  - you may change the default behavior by setting twoTrimLeftEnumSets in the TTextWriter.CustomOptions property of a given serializer
  - note that unserialization process would recognize both formats

procedure SetText(var result: RawUTF8; reformat: TTextWriterJSONFormat=jsonCompact);

  Retrieve the data as a string
  - will avoid creation of a temporary RawUTF8 variable as for Text function

class procedure UnRegisterCustomJSONSerializer(aTypeInfo: pointer);

  Undefine a custom serialization for a given dynamic array or record
  - it will un-register any callback or text-based custom serialization i.e. any previous RegisterCustomJSONSerializer() or RegisterCustomJSONSerializerFromText() call
  - expects TypeInfo() from a dynamic array or a record (will raise an exception otherwise)
  - it will set back to the default binary + Base64 encoding serialization

procedure WrBase64(P: PAnsiChar; Len: PtrUInt; withMagic: boolean);

  Write some data Base64 encoded
  - if withMagic is TRUE, will write as "\uFFF0base64encodedbinary"

procedure WriteObject(Value: TObject; Options: TTextWriterWriteObjectOptions=[woDontStoreDefault]); virtual;

  Serialize as JSON the given object
  - this default implementation will write null, or only write the class name and pointer if FullExpand is true - use TJSONSerializer.WriteObject method for full RTTI handling
  - default implementation will write TList/TCollection/TStrings/TRawUTF8List as appropriate array of class name/pointer (if woFullExpand is set)

procedure WriteObjectAsString(Value: TObject; Options: TTextWriterWriteObjectOptions=[woDontStoreDefault]);

  Same as WriteObject(), but will double all internal " and bound with "
  - this implementation will avoid most memory allocations

procedure WrRecord(const Rec; TypeInfo: pointer);

  Write some record content as binary, Base64 encoded with our magic prefix

property CustomOptions: TTextWriterOptions read fCustomOptions write fCustomOptions;

  Global options to customize this TTextWriter instance process
  - allows to override e.g. AddRecordJSON() and AddDynArrayJSON() behavior

property OnFlushToStream: TOnTextWriterFlush read fOnFlushToStream write fOnFlushToStream;

  Optional event called before FlushToStream method process
property OnWriteObject: TOnTextWriterObjectProp read fOnWriteObject write
fOnWriteObject;

    Allows to override default WriteObject property JSON serialization

property Stream: TStream read fStream write SetStream;

    The internal TStream used for storage
    - you should call the FlushFinal (or FlushToStream) methods before using this TStream content,
      to flush all pending characters
    - if the TStream instance has not been specified when calling the TTextWriter constructor, it can
      be forced via this property, before any writing

property TextLength: PtrUInt read GetTextLength;

    Count of added bytes to the stream
    - see PendingBytes for the number of bytes currently in the memory buffer or WrittenBytes for
      the number of bytes already written to disk

property WrittenBytes: PtrUInt read fTotalFileSize;

    How many bytes were currently written on disk
    - excluding the bytes in the internal buffer
    - see TextLength for the total number of bytes, on both disk and memory

TTextWriterWithEcho = class(TTextWriter)

    Stream TEXT writer, with optional echoing of the lines
    - as used e.g. by TSynLog writer for log optional redirection
    - is defined as a sub-class to reduce plain TTextWriter scope
    - see SynTable.pas for SQL resultset export via TJSOWriter
    - see mORMot.pas for proper class serialization via TJSONSerializer.writeObject

procedure AddEndOfLine(aLevel: TSynLogInfo=sllNone);

    Mark an end of line, ready to be "echoed" to registered listeners
    - append a LF (#10) char or CR+LF (#13#10) chars to the buffer, depending on the EndOfLineCRLF
      property value (default is LF, to minimize storage)
    - any callback registered via EchoAdd() will monitor this line
    - used e.g. by TSynLog for console output, as stated by Level parameter

procedure EchoAdd(const aEcho: TOnTextWriterEcho);

    Add a callback to echo each line written by this class
    - this class expects AddEndOfLine to mark the end of each line

procedure EchoRemove(const aEcho: TOnTextWriterEcho);

    Remove a callback to echo each line written by this class
    - event should have been previously registered by a EchoAdd() call

procedure EchoReset;

    Reset the internal buffer used for echoing content

procedure FlushToStream; override;

    Write pending data to the Stream, with automatic buffer resizal and echoing
    - this overridden method will handle proper echoing
property EndOfLineCRLF: boolean read GetEndOfLineCRLF write SetEndOfLineCRLF;
  Define how AddEndOfLine method stores its line feed characters
  - by default (FALSE), it will append a LF (#10) char to the buffer
  - you can set this property to TRUE, so that CR+LF (#13#10) chars will be appended instead
  - is just a wrapper around twoEndOfLineCRLF item in CustomOptions

TObjectListHashedAbstract = class(TObject)
  Abstract ancestor to manage a dynamic array of TObject
  - do not use this abstract class directly, but rather the inherited TObjectListHashed and TObjectListPropertyHashed

constructor Create(aFreeItems: boolean=true); reintroduce;
  Initialize the class instance
  - if aFreeItems is TRUE (default), will behave like a TObjectList
  - if aFreeItems is FALSE, will behave like a TList

destructor Destroy; override;
  Release used memory

function Add(aObject: TObject; out wasAdded: boolean): integer; virtual; abstract;
  Search and add an object reference to the list
  - returns the found/added index

function IndexOf(aObject: TObject): integer; virtual; abstract;
  Retrieve an object index within the list, using a fast hash table
  - returns -1 if not found

procedure Delete(aObject: TObject); overload; virtual;
  Delete an object from the list
  - will invalidate the whole hash table

procedure Delete(aIndex: integer); overload;
  Delete an object from the list
  - the internal hash table is not recreated, just invalidated (i.e. this method calls HashInvalidate not FindHashedAndDelete)
  - will invalidate the whole hash table

property Count: integer read fCount;
  Returns the count of stored objects

property Hash: TDynArrayHashed read fHash;
  Direct access to the underlying hashing engine

property List: TObjectDynArray read fList;
  Direct access to the items list array

TObjectListHashed = class(TObjectListHashedAbstract)
  This class behaves like TList/TObjectList, but will use hashing for (much) faster IndexOf() method
function Add(aObject: TObject; out wasAdded: boolean): integer; override;
  Search and add an object reference to the list
  - returns the found/added index
  - if added, hash is stored and Items[] := aObject

function IndexOf(aObject: TObject): integer; override;
  Retrieve an object index within the list, using a fast hash table
  - returns -1 if not found

procedure Delete(aObject: TObject); override;
  Delete an object from the list
  - overridden method won’t invalidate the whole hash table, but refresh it

TObjectListPropertyHashed = class(TObjectListHashedAbstract)
  This class will hash and search for a sub property of the stored objects

constructor Create(aSubPropAccess: TObjectListPropertyHashedAccessProp;
aHashCode: TDynArrayHashOne=nil; aCompare: TDynArraySortCompare=nil;
aFreeItems: boolean=true); reintroduce;
  Initialize the class instance with the corresponding callback in order to handle sub-property hashing and search
  - see TSetWeakZeroClass in mORMot.pas unit as example:
    function WeakZeroClassSubProp(aObject: TObject): TObject;
    begin
      result := TSetWeakZeroInstance(aObject).fInstance;
    end;
  - by default, aHashCode/aCompare will hash/search for pointers: you can specify the hash/search methods according to your sub property (e.g. HashAnsiStringI/SortDynArrayAnsiStringI for a RawUTF8)
  - if aFreeItems is TRUE (default), will behave like a TObjectList; if aFreeItems is FALSE, will behave like a TList

function Add(aObject: TObject; out wasAdded: boolean): integer; override;
  Search and add an object reference to the list
  - returns the found/added index
  - if added, only the hash is stored: caller has to set List[i]

function IndexOf(aObject: TObject): integer; override;
  Retrieve an object index within the list, using a fast hash table
  - returns -1 if not found

TPointerClassHashed = class(TObject)
  Abstract class stored by a TPointerClassHash list

constructor Create(aInfo: pointer);
  Initialize the instance

property Info: pointer read fInfo write fInfo;
  The associated information of this instance
  - may be e.g. a PTypeInfo value, when caching RTTI information
TPointerClassHash = class(TObjectListPropertyHashed)

Handle a O(1) hashed-based storage of TPointerClassHashed, from a pointer
- used e.g. to store RTTI information from its PTypeInfo value
- if not thread safe, but could be used to store RTTI, since all type information should have been initialized before actual process

constructor Create;
  Initialize the storage list

function Find(aInfo: pointer): TPointerClassHashed;
  Search for a stored instance, from its supplied pointer reference
  - returns nil if aInfo was not previously added by FindOrAdd()
  - this method is not thread-safe

function TryAdd(aInfo: pointer): PPointerClassHashed;
  Try to add an entry to the storage
  - returns nil if the supplied information is already in the list
  - returns a pointer to where a newly created TPointerClassHashed instance should be stored
  - this method is not thread-safe

TPointerClassHashLocked = class(TPointerClassHash)

Handle a O(1) hashed-based storage of TPointerClassHashed, from a pointer
- this inherited class add a mutex to be thread-safe

constructor Create;
  Initialize the storage list

destructor Destroy; override;
  Finalize the storage list

function FindLocked(aInfo: pointer): TPointerClassHashed;
  Search for a stored instance, from its supplied pointer reference
  - returns nil if aInfo was not previously added by FindOrAdd()
  - this overridden method is thread-safe, unless returned TPointerClassHashed instance is deleted in-between
function TryAddLocked(aInfo: pointer; out aNewEntry: PPointerClassHashed): boolean;
  
  *Try to add an entry to the storage*
  - returns false if the supplied information is already in the list
  - returns true, and a pointer to where a newly created TPointerClassHashed instance should be stored: in this case, you should call UnLock once set
  - could be used as such:

  ```pascal
  var entry: PPointerClassHashed;
  ...
  if HashList.TryAddLocked(aTypeInfo, entry) then
    try
      entry^ := TMyCustomPointerClassHashed.Create(aTypeInfo, ...);
    finally
      HashList.Unlock;
    end;
  ...
  ```

procedure Unlock;
  
  *Release the lock after a previous TryAddLocked()=true call* 

TSynObjectListLocked = class(TSynObjectList)
  
  *Add locking methods to a TSynObjectList*
  - this class overrides the regular TSynObjectList, and do not share any code with the TObjectListHashedAbstract/TObjectListHashed classes
  - you need to call the Safe.Lock/Unlock methods by hand to protect the execution of index-oriented methods (like Delete/Items/Count...): the list content may change in the background, so using indexes is thread-safe
  - on the other hand, Add/Clear/ClearFromLast/Remove stateless methods have been overridden in this class to call Safe.Lock/Unlock, and therefore are thread-safe and protected to any background change

constructor Create(aOwnsObjects: boolean=true); reintroduce;
  
  *Initialize the list instance*
  - the stored TObject instances will be owned by this TSynObjectListLockeded, unless AOwnsObjects is set to false

destructor Destroy; override;
  
  *Release the list instance (including the locking resource)*

function Add(item: pointer): integer; override;
  
  *Add one item to the list using the global critical section*

function Exists(item: pointer): boolean; override;
  
  *Check an item using the global critical section*

function Remove(item: pointer): integer; override;
  
  *Fast delete one item in the list*

procedure Clear; override;
  
  *Delete all items of the list using the global critical section*

procedure ClearFromLast; override;
  
  *Delete all items of the list in reverse order, using the global critical section*
property Safe: TSynLocker read fSafe;

The critical section associated to this list instance
- could be used to protect shared resources within the internal process, for index-oriented
  methods like Delete/Items/Count...
- use Safe.Lock/TryLock with a try ... finally Safe.Unlock block

TRawUTF8List = class(TObject)

TStringList-class optimized to work with our native UTF-8 string type
- can optionally store associated some TObject instances
- high-level methods of this class are thread-safe
- if fNoDuplicate flag is defined, an internal hash table will be maintained to perform IndexOf()
  lookups in O(1) linear way

constructor Create(aOwnObjects: boolean; aNoDuplicate: boolean=false;
aCaseSensitive: boolean=true); overload;

  Backward compatibility overloaded constructor
  - please rather use the overloaded Create(TRawUTF8ListFlags)

constructor Create(aFlags: TRawUTF8ListFlags=[fCaseSensitive]); overload;

  Initialize the RawUTF8/Objects storage
  - by default, any associated Objects[] are just weak references; you may supply fOwnObjects flag
to force object instance management
  - if you want the stored text items to be unique, set fNoDuplicate and then an internal hash
table will be maintained for fast IndexOf()
  - you can unset fCaseSensitive to let the UTF-8 lookup be case-insensitive

destructor Destroy; override;

  Finalize the internal objects stored
  - if instance was created with fOwnObjects flag

function Add(const aText: RawUTF8; aRaiseExceptionIfExisting: boolean=false): PtrInt;

  Store a new RawUTF8 item
  - without the fNoDuplicate flag, it will always add the supplied value
  - if fNoDuplicate was set and aText already exists (using the internal hash table), it will return -1
    unless aRaiseExceptionIfExisting is forced
  - thread-safe method

function AddObject(const aText: RawUTF8; aObject: TObject;
aRaiseExceptionIfExisting: boolean=false; aFreeAndReturnExistingObject:
PPointer=nil): PtrInt;

  Store a new RawUTF8 item, and its associated TObject
  - without the fNoDuplicate flag, it will always add the supplied value
  - if fNoDuplicate was set and aText already exists (using the internal hash table), it will return -1
    unless aRaiseExceptionIfExisting is forced; optionally freeing the supplied aObject if
    aFreeAndReturnExistingObject is true, in which pointer the existing Objects[] is copied (see
    AddObjectUnique as a convenient wrapper around this behavior)
  - thread-safe method
function Contains(const aText: RawUTF8; aFirstIndex: integer=0): PtrInt;
    
    Search for any RawUTF8 item containing some text
    - uses PosEx() on the stored lines
    - this method is not thread-safe since the internal list may change and the returned index may
      not be accurate any more
    - by design, aText lookup can't use the internal Hash Table

function Delete(const aText: RawUTF8): PtrInt; overload;
    
    Delete a stored RawUTF8 item, and its associated TObject
    - will search for the value using IndexOf(aText), and returns its index
    - returns -1 if no entry was found and deleted
    - thread-safe method, using the internal Hash Table if fNoDuplicate is set

function DeleteFromName(const Name: RawUTF8): PtrInt;
    virtual;
    
    Delete a stored RawUTF8 item, and its associated TObject, from a given Name when stored as
    'Name=Value' pairs
    - raise no exception in case of out of range supplied index
    - thread-safe method, but not using the internal Hash Table
    - consider using TSynNameValue if you expect efficient name/value process

function GetObjectFrom(const aText: RawUTF8): pointer;
    
    Get a stored Object item by its associated UTF8 text
    - returns nil and raise no exception if aText doesn't exist
    - thread-safe method, unless returned TObject is deleted in the background

function GetText(const Delimiter: RawUTF8=#13#10): RawUTF8;
    
    Retrieve the all lines, separated by the supplied delimiter
    - this method is thread-safe

function GetValueAt(Index: PtrInt): RawUTF8;
    
    Access to the Value of a given 'Name=Value' pair at a given position
    - this method is not thread-safe
    - consider using TSynNameValue if you expect efficient name/value process

function IndexOf(const aText: RawUTF8): PtrInt;
    
    Find a RawUTF8 item in the stored Strings[] list
    - this search is case sensitive if fCaseSensitive flag was set (which is the default)
    - this method is not thread-safe since the internal list may change and the returned index may
      not be accurate any more
    - see also GetObjectFrom()
    - uses the internal Hash Table if fNoDuplicate was set

function IndexOfName(const Name: RawUTF8): PtrInt;
    
    Find the index of a given Name when stored as 'Name=Value' pairs
    - search on Name is case-insensitive with 'Name=Value' pairs
    - this method is not thread-safe, and won't use the internal Hash Table
    - consider using TSynNameValue if you expect efficient name/value process
function IndexOfObject(aObject: TObject): PtrInt;

Find a TObject item index in the stored Objects[] list
- this method is not thread-safe since the internal list may change and the returned index may not be accurate any more
- aObject lookup won't use the internal Hash Table

function PopFirst(out aText: RawUTF8; aObject: PObject=nil): boolean;

Retrieve and delete the first RawUTF8 item in the list
- could be used as a FIFO, calling Add() as a "push" method
- thread-safe method

function PopLast(out aText: RawUTF8; aObject: PObject=nil): boolean;

Retrieve and delete the last RawUTF8 item in the list
- could be used as a FILO, calling Add() as a "push" method
- thread-safe method

function UpdateValue(const Name: RawUTF8; var Value: RawUTF8; ThenDelete: boolean): boolean;

Retrieve Value from an existing Name=Value, then optionally delete the entry
- if Name is found, will fill Value with the stored content and return true
- if Name is not found, Value is not modified, and false is returned
- thread-safe method, but not using the internal Hash Table
- consider using TSynNameValue if you expect efficient name/value process

procedure AddObjectUnique(const aText: RawUTF8; aObjectToAddOrFree: PPointer);

Try to store a new RawUTF8 item and its associated TObject
- fNoDuplicate should have been specified in the list flags
- if aText doesn't exist, will add the values
- if aText exist, will call aObjectToAddOrFree.Free and set the value already stored in Objects[]
into aObjectToAddOrFree - allowing dual commit thread-safe update of the list, e.g. after a previous unsuccessful call to GetObjectFrom(aText)
- thread-safe method, using an internal Hash Table to speedup IndexOf()
- in this method, this method is just a wrapper around
  AddObject(aText,aObjectToAddOrFree^,false,aObjectToAddOrFree);

procedure AddRawUTF8List(List: TRawUTF8List);

Append a specified list to the current content
- thread-safe method

procedure BeginUpdate;

The OnChange event will be raised only when EndUpdate will be called
- this method will also call Safe.Lock for thread-safety

procedure Clear; virtual;

Erase all stored RawUTF8 items
- and corresponding objects (if aOwnObjects was true at constructor)
- thread-safe method, also clearing the internal Hash Table

procedure Delete(Index: PtrInt); overload;

Delete a stored RawUTF8 item, and its associated TObject
- raise no exception in case of out of range supplied index
- this method is not thread-safe: use Safe.Lock/UnLock if needed
procedure EndUpdate;
      Call theOnChange event if changes occurred
      - this method will also call Safe.UnLock for thread-safety

procedure LoadFromFile(const FileName: TFileName);
      Set all lines from an UTF-8 text file
      - expect the file is explicitly an UTF-8 file
      - will ignore any trailing UTF-8 BOM in the file content, but will not expect one either
      - this method is thread-safe

procedure SaveToFile(const FileName: TFileName; const Delimiter: RawUTF8=#13#10);
      Write all lines into a new file
      - this method is thread-safe

procedure SaveToStream(Dest: TStream; const Delimiter: RawUTF8=#13#10);
      Write all lines into the supplied stream
      - this method is thread-safe

procedure SetFrom(const aText: TRawUTF8DynArray; const aObject: TObjectDynArray);
      Set low-level text and objects from existing arrays

procedureSetText(const aText: RawUTF8; const Delimiter: RawUTF8=#13#10);
      Set all lines, separated by the supplied delimiter
      - this method is thread-safe

property Capacity: PtrInt read GetCapacity write SetCapacity;
      Set or retrieve the current memory capacity of the RawUTF8 list
      - reading this property is not thread-safe, since size may change

property CaseSensitive: boolean read GetCaseSensitive write SetCaseSensitive;
      Set if IndexOf() shall be case sensitive or not
      - default is TRUE
      - matches fCaseSensitive in Flags

property Count: PtrInt read GetCount;
      Return the count of stored RawUTF8
      - reading this property is not thread-safe, since size may change

property Flags: TRawUTF8ListFlags read fFlags write fFlags;
      Access to the low-level flags of this list

property Names[Index: PtrInt]: RawUTF8 read GetName;
      Retrieve the corresponding Name when stored as 'Name=Value' pairs
      - reading this property is not thread-safe, since content may change
      - consider TSynNameValue if you expect more efficient name/value process

property NameValueSep: AnsiChar read fNameValueSep write fNameValueSep;
      The char separator between 'Name=Value' pairs
      - equals '=' by default
      - consider TSynNameValue if you expect more efficient name/value process
property NoDuplicate: boolean read GetNoDuplicate;

Set if the list doesn't allow duplicated UTF-8 text
- if true, an internal hash table is maintained for faster IndexOf()
- matches fNoDuplicate in Flags

property ObjectPtr: PPointerArray read GetObjectPtr;

Direct access to the memory of the TObjectDynArray items
- reading this property is not thread-safe, since content may change

property Objects[Index: PtrInt]: pointer read GetObject write PutObject;

Get or set a Object item
- returns nil and raise no exception in case of out of range supplied index
- reading this property is not thread-safe, since content may change

property OnChange: TNotifyEvent read fOnChange write fOnChange;

Event triggered when an entry is modified

property Safe: TSynLocker read fSafe;

Access to the locking methods of this instance
- use Safe.Lock/TryLock with a try ... finally Safe.Unlock block

property Strings[Index: PtrInt]: RawUTF8 read Get write Put;

Get or set a RawUTF8 item
- returns "" and raise no exception in case of out of range supplied index
- if you want to use it with the VCL, use UTF8ToString() function
- reading this property is not thread-safe, since content may change

property Text: RawUTF8 read GetTextCRLF write SetTextCRLF;

Set or retrieve all items as text lines
- lines are separated by #13#10 (CRLF) by default; use GetText and SetText methods if you want to use another line delimiter (even a comma)
- this property is thread-safe

property TextPtr: PPUtf8CharArray read GetTextPtr;

Direct access to the memory of the TRawUTF8DynArray items
- reading this property is not thread-safe, since content may change

property Values[const Name: RawUTF8]: RawUTF8 read GetValue write SetValue;

Access to the corresponding 'Name=Value' pairs
- search on Name is case-insensitive with 'Name=Value' pairs
- reading this property is thread-safe, but won't use the hash table
- consider TSynNameValue if you expect more efficient name/value process

property ValuesArray: TDynArrayHashed read fValues;

Direct access to the TRawUTF8DynArray items dynamic array wrapper
- using this property is not thread-safe, since content may change

TAlgoCompress = class(TSynPersistent)

Abstract low-level parent class for generic compression/decompression algorithms
- will encapsulate the compression algorithm with crc32c hashing
- all Algo* abstract methods should be overridden by inherited classes
constructor Create; override;

  Will register AlgoID in the global list, for Algo() class methods
  - no need to free this instance, since it will be owned by the global list
  - raise a ESynException if the class or its AlgoID are already registered
  - you should never have to call this constructor, but define a global variable holding a reference
to a shared instance

class function Algo(Comp: PAnsiChar; CompLen: integer; out IsStored: boolean): TAlgoCompress; overload;
  Get the TAlgoCompress instance corresponding to the AlgoID stored in the supplied compressed
buffer
  - returns nil if no algorithm was identified
  - also identifies "stored" content in IsStored variable

class function Algo(const Comp: RawByteString): TAlgoCompress; overload;
  Get the TAlgoCompress instance corresponding to the AlgoID stored in the supplied compressed
buffer
  - returns nil if no algorithm was identified

class function Algo(Comp: PAnsiChar; CompLen: integer): TAlgoCompress; overload;
  Get the TAlgoCompress instance corresponding to the AlgoID stored in the supplied compressed
buffer
  - returns nil if no algorithm was identified

class function Algo(const Comp: TByteDynArray): TAlgoCompress; overload;
  Get the TAlgoCompress instance corresponding to the AlgoID stored in the supplied compressed
buffer
  - returns nil if no algorithm was identified

class function Algo(AlgoID: byte): TAlgoCompress; overload;
  Get the TAlgoCompress instance corresponding to the supplied AlgoID
  - returns nil if no algorithm was identified
  - stored content is identified as TAlgoSynLZ

function AlgoCompress(Plain: pointer; PlainLen: integer; Comp: pointer): integer;
  virtual; abstract;
  This method will compress the supplied data

function AlgoCompressDestLen(PlainLen: integer): integer; virtual; abstract;
  Get maximum possible (worse) compressed size for the supplied length

function AlgoDecompress(Comp: pointer; CompLen: integer; Plain: pointer): integer;
  virtual; abstract;
  This method will decompress the supplied data

function AlgoDecompressDestLen(Comp: pointer): integer; virtual; abstract;
  This method will return the size of the decompressed data

function AlgoDecompressPartial(Comp: pointer; CompLen: integer; Partial: pointer; PartialLen, PartialLenMax: integer): integer; virtual; abstract;
  This method will partially and safely decompress the supplied data
  - expects PartialLen <= result < PartialLenMax, depending on the algorithm
function AlgoHash(Previous: cardinal; Data: pointer; DataLen: integer): cardinal;
    virtual;
    Computes by default the crc32c() digital signature of the buffer

function AlgoID: byte; virtual; abstract;
    Should return a genuine byte identifier
    - 0 is reserved for stored, 1 for TAlgoSynLz, 2/3 for TAlgoDeflate/Fast (in mORMot.pas), 4/5/6 for TAlgoLizard/Fast/Huffman (in SynLizard.pas)

function AlgoName: TShort16;
    Returns the algorithm name, from its classname
    - e.g. TAlgoSynLZ -> 'synlz' TAlgoLizard -> 'lizard' nil -> 'none'

function Compress(Plain: PAnsiChar; PlainLen: integer; CompressionSizeTrigger: integer=100; CheckMagicForCompressed: boolean=false; BufferOffset: integer=0): RawByteString; overload;
    Compress a memory buffer with crc32c hashing to a RawByteString

function Compress(const Plain: RawByteString; CompressionSizeTrigger: integer=100; CheckMagicForCompressed: boolean=false; BufferOffset: integer=0): RawByteString; overload;
    Compress a memory buffer with crc32c hashing to a RawByteString

function Compress(Plain, Comp: PAnsiChar; PlainLen, CompLen: integer; CompressionSizeTrigger: integer=100; CheckMagicForCompressed: boolean=false): integer; overload;
    Compress a memory buffer with crc32c hashing
    - supplied Comp buffer should contain at least CompressDestLen(PlainLen) bytes

function CompressDestLen(PlainLen: integer): integer;
    Get maximum possible (worse) compressed size for the supplied length
    - including the crc32c + algo 9 bytes header

function CompressToBytes(Plain: PAnsiChar; PlainLen: integer; CompressionSizeTrigger: integer=100; CheckMagicForCompressed: boolean=false): TByteDynArray; overload;
    Compress a memory buffer with crc32c hashing to a TByteDynArray

function CompressToBytes(const Plain: RawByteString; CompressionSizeTrigger: integer=100; CheckMagicForCompressed: boolean=false): TByteDynArray; overload;
    Compress a memory buffer with crc32c hashing to a TByteDynArray

function Decompress(const Comp: RawByteString; out PlainLen: integer; var tmp: RawByteString; Load: TAlgoCompressLoad=aclNormal): pointer; overload;
    Uncompress a RawByteString memory buffer with crc32c hashing
    - returns nil if crc32 hash failed, i.e. if the supplied Comp is not correct
    - returns a pointer to the uncompressed data and fill PlainLen variable, after crc32c hash
    - avoid any memory allocation in case of a stored content - otherwise, would uncompress to the tmp variable, and return pointer(tmp) and length(tmp)

function Decompress(const Comp: TByteDynArray): RawByteString; overload;
    Uncompress a RawByteString memory buffer with crc32c hashing
function Decompress(const Comp: RawByteString; Load: TAlgoCompressLoad=aclNormal; BufferOffset: integer=0): RawByteString; overload;
  Uncompress a RawByteString memory buffer with crc32c hashing

function Decompress(Comp: PAnsiChar; CompLen: integer; out PlainLen: integer; var tmp: RawByteString; Load: TAlgoCompressLoad=aclNormal): pointer; overload;
  Uncompress a RawByteString memory buffer with crc32c hashing
  - returns nil if crc32 hash failed, i.e. if the supplied Data is not correct
  - returns a pointer to an uncompressed data buffer of PlainLen bytes
  - avoid any memory allocation in case of a stored content - otherwise, would uncompress to the tmp variable, and return pointer(tmp) and length(tmp)

function DecompressBody(Comp, Plain: PAnsiChar; CompLen, PlainLen: integer; Load: TAlgoCompressLoad=aclNormal): boolean;
  Decode the content of a memory buffer compressed via the Compress() method
  - PlainLen has been returned by a previous call to DecompressHeader()

function DecompressHeader(Comp: PAnsiChar; CompLen: integer; Load: TAlgoCompressLoad=aclNormal): integer;
  Decode the header of a memory buffer compressed via the Compress() method
  - validates the crc32c of the compressed data (unless Load=aclNoCrcFast), then return the uncompressed size in bytes, or 0 if the crc32c does not match
  - should call DecompressBody() later on to actually retrieve the content

function DecompressPartial(Comp, Partial: PAnsiChar; CompLen, PartialLen, PartialLenMax: integer): integer;
  Partial decoding of a memory buffer compressed via the Compress() method
  - returns 0 on error, or how many bytes have been written to Partial
  - will call virtual AlgoDecompressPartial() which is slower, but expected to avoid any buffer overflow on the Partial destination buffer
  - some algorithms (e.g. Lizard) may need some additional bytes in the decode buffer, so PartialLenMax bytes should be allocated in Partial^, with PartialLenMax > expected PartialLen, and returned bytes may be > PartialLen, but always <= PartialLenMax

function TryDecompress(const Comp: RawByteString; out Dest: RawByteString; Load: TAlgoCompressLoad=aclNormal): boolean;
  Uncompress a RawByteString memory buffer with crc32c hashing
  - returns TRUE on success

class function UncompressedSize(const Comp: RawByteString): integer;
  Quickly validate a compressed buffer content, without uncompression
  - extract the TAlgoCompress, and call DecompressHeader() to check the hash of the compressed data, and return then uncompressed size
  - returns 0 on error (e.g. unknown algorithm or incorrect hash)

procedure Decompress(Comp: PAnsiChar; CompLen: integer; out Result: RawByteString; Load: TAlgoCompressLoad=aclNormal; BufferOffset: integer=0); overload;
  Uncompress a memory buffer with crc32c hashing
TAlgoSynLZ = class(TAlgoCompress)

Implement our fast SynLZ compression as a TAlgoCompress class
- please use the AlgoSynLZ global variable methods instead of the deprecated SynLZCompress/SynLZDecompress wrapper functions

function AlgoCompress(Plain: pointer; PlainLen: integer; Comp: pointer): integer; override;
   Compress the supplied data using SynLZ

function AlgoCompressDestLen(PlainLen: integer): integer; override;
   Get maximum possible (worse) SynLZ compressed size for the supplied length

function AlgoDecompress(Comp: pointer; CompLen: integer; Plain: pointer): integer; override;
   Decompress the supplied data using SynLZ

function AlgoDecompressDestLen(Comp: pointer): integer; override;
   Return the size of the SynLZ decompressed data

function AlgoDecompressPartial(Comp: pointer; CompLen: integer; Partial: pointer;
   PartialLen, PartialLenMax: integer): integer; override;
   Partial (and safe) decompression of the supplied data using SynLZ

function AlgoID: byte; override;
   Returns 1 as genuine byte identifier for SynLZ

TAlgoCompressWithNoDestLen = class(TAlgoCompress)

Abstract class storing the plain length before calling compression API
- some libraries (e.g. Deflate or Lizard) don't provide the uncompressed length from its output buffer - inherit from this class to store this value as ToVarUInt32, and override the RawProcess abstract protected method

function AlgoCompress(Plain: pointer; PlainLen: integer; Comp: pointer): integer; override;
   Performs the compression, storing PlainLen and calling protected RawProcess

function AlgoDecompress(Comp: pointer; CompLen: integer; Plain: pointer): integer; override;
   Performs the decompression, retrieving PlainLen and calling protected RawProcess

function AlgoDecompressDestLen(Comp: pointer): integer; override;
   Return the size of the decompressed data (using FromVarUInt32)

function AlgoDecompressPartial(Comp: pointer; CompLen: integer; Partial: pointer;
   PartialLen, PartialLenMax: integer): integer; override;
   Performs the decompression, retrieving PlainLen and calling protected RawProcess
TSynDictionary = class(TSynPersistentLock)

Thread-safe dictionary to store some values from associated keys
- will maintain a dynamic array of values, associated with a hash table for the keys, so that setting or retrieving values would be O(1)
- all process is protected by a TSynLocker, so will be thread-safe
- TDynArray is a wrapper which do not store anything, whereas this class is able to store both keys and values, and provide convenient methods to access the stored data, including JSON serialization and binary storage

constructor Create(aKeyTypeTypeInfo,aValueTypeInfo: pointer; aKeyCaseInsensitive: boolean=false; aTimeoutSeconds: cardinal=0; aCompressAlgo: TAlgoCompress=nil);
reintroduce; virtual;
Initialize the dictionary storage, specifying dynamic array keys/values
- aKeyTypeTypeInfo should be a dynamic array TypeInfo() RTTI pointer, which would store the keys within this TSynDictionary instance
- aValueTypeInfo should be a dynamic array TypeInfo() RTTI pointer, which would store the values within this TSynDictionary instance
- by default, string keys would be searched following exact case, unless aKeyCaseInsensitive is TRUE
- you can set an optional timeout period, in seconds - you should call DeleteDeprecated periodically to search for deprecated items

destructor Destroy; override;
Finalize the storage
- would release all internal stored values

function Add(const aKey, aValue): integer;
Try to add a value associated with a primary key
- returns the index of the inserted item, -1 if aKey is already existing
- this method is thread-safe, since it will lock the instance

function AddInArray(const aKey, aArrayValue): boolean;
Add aArrayValue item within a dynamic-array value associated via aKey
- expect the stored value to be a dynamic array itself
- would search for aKey as primary key, then use TDynArray.Add to add aArrayValue to the associated dynamic array
- returns FALSE if Values is not a tkDynArray, or if aKey was not found
- this method is thread-safe, since it will lock the instance

function AddOnceInArray(const aKey, aArrayValue): boolean;
Add once aArrayValue within a dynamic-array value associated via aKey
- expect the stored value to be a dynamic array itself
- would search for aKey as primary key, then use TDynArray.FindAndAddIfNotExisting to add once aArrayValue to the associated dynamic array
- returns FALSE if Values is not a tkDynArray, or if aKey was not found
- this method is thread-safe, since it will lock the instance
function AddOrUpdate(const aKey, aValue): integer;

Store a value associated with a primary key
- returns the index of the matching item
- if aKey does not exist, a new entry is added
- if aKey does exist, the existing entry is overridden with aValue
- this method is thread-safe, since it will lock the instance

function Clear(const aKey): integer;

Clear the value associated via aKey
- does not delete the entry, but reset its value
- returns the index of the matching item, -1 if aKey was not found
- this method is thread-safe, since it will lock the instance

function Count: integer;

Returns how many items are currently stored in this dictionary
- this method is thread-safe

function Delete(const aKey): integer;

Delete a key/value association from its supplied aKey
- this would delete the entry, i.e. matching key and value pair
- returns the index of the deleted item, -1 if aKey was not found
- this method is thread-safe, since it will lock the instance

function DeleteAt(aIndex: integer): boolean;

Delete a key/value association from its internal index
- this method is not thread-safe: you should use fSafe.Lock/Unlock e.g. then Find/FindValue to retrieve the index value

function DeleteDeprecated: integer;

Search and delete all deprecated items according to TimeoutSeconds
- returns how many items have been deleted
- you can call this method very often: it will ensure that the search process will take place at most once every second
- this method is thread-safe, but blocking during the process

function DeleteInArray(const aKey, aArrayValue): boolean;

Clear aArrayValue item of a dynamic-array value associated via aKey
- expect the stored value to be a dynamic array itself
- would search for aKey as primary key, then use TDynArray.FindAndDelete to delete any aArrayValue match in the associated dynamic array
- returns FALSE if Values is not a TDynArray, or if aKey or aArrayValue were not found
- this method is thread-safe, since it will lock the instance

function Exists(const aKey): boolean;

Search for a primary key presence
- returns TRUE if aKey was found, FALSE if no match exists
- this method is thread-safe
function Find(const aKey; aUpdateTimeOut: boolean=false): integer;
  Search of a primary key within the internal hashed dictionary
  - returns the index of the matching item, -1 if aKey was not found
  - if you want to access the value, you should use fSafe.Lock/Unlock: consider using Exists or
    FindAndCopy thread-safe methods instead
  - aUpdateTimeOut will update the associated timeout value of the entry

function FindAndCopy(const aKey; out aValue; aUpdateTimeOut: boolean=true): boolean;
  Search of a stored value by its primary key, and return a local copy
  - so this method is thread-safe
  - returns TRUE if aKey was found, FALSE if no match exists
  - will update the associated timeout value of the entry, unless aUpdateTimeOut is set to false

function FindAndExtract(const aKey; out aValue): boolean;
  Search of a stored value by its primary key, then delete and return it
  - returns TRUE if aKey was found, fill aValue with its content, and delete the entry in the internal
    storage
  - so this method is thread-safe
  - returns FALSE if no match exists

function FindInArray(const aKey, aArrayValue): boolean;
  Search aArrayValue item in a dynamic-array value associated via aKey
  - expect the stored value to be a dynamic array itself
  - would search for aKey as primary key, then use TDynArray.Find to delete any aArrayValue
    match in the associated dynamic array
  - returns FALSE if Values is not a tdynArray, or if aKey or aArrayValue were not found
  - this method is thread-safe, since it will lock the instance

function FindKeyFromValue(const aValue; out aKey; aUpdateTimeOut: boolean=true): boolean;
  Search of a stored key by its associated key, and return a key local copy
  - won't use any hashed index but TDynArray.IndexOf over fValues, so is much slower than
    FindAndCopy()
  - will update the associated timeout value of the entry, unless aUpdateTimeOut is set to false
  - so this method is thread-safe
  - returns TRUE if aValue was found, FALSE if no match exists

function FindValue(const aKey; aUpdateTimeOut: boolean=false; aIndex: PInteger=nil): pointer;
  Search of a primary key within the internal hashed dictionary
  - returns a pointer to the matching item, nil if aKey was not found
  - if you want to access the value, you should use fSafe.Lock/Unlock: consider using Exists or
    FindAndCopy thread-safe methods instead
  - aUpdateTimeOut will update the associated timeout value of the entry
function FindValueOrAdd(const aKey; var added: boolean; aIndex: PInteger=nil): pointer;
    \textit{Search of a primary key within the internal hashed dictionary}
    - returns a pointer to the matching or already existing item
    - if you want to access the value, you should use fSafe.Lock/Unlock: consider using Exists or
      FindAndCopy thread-safe methods instead
    - will update the associated timeout value of the entry, if applying

function ForEach(const OnMatch: TSynDictionaryEvent; KeyCompare,ValueCompare:
  TDynArraySortCompare; const aKey,aValue; Opaque: pointer=nil): integer; overload;
    \textit{Apply a specified event over matching items stored in this dictionary}
    - would browse the list in the adding order, comparing each key and/or value item with the
      supplied comparison functions and aKey/aValue content
    - returns the number of times OnMatch has been called, i.e. how many times
      KeyCompare(aKey,Keys[#])=0 or ValueCompare(aValue,Values[#])=0
    - this method is thread-safe, since it will lock the instance

function ForEach(const OnEach: TSynDictionaryEvent; Opaque: pointer=nil): integer;
    \textit{Apply a specified event over all items stored in this dictionary}
    - would browse the list in the adding order
    - returns the number of times OnEach has been called
    - this method is thread-safe, since it will lock the instance

function LoadFromBinary(const binary: RawByteString): boolean;
    \textit{Load the content from SynLZ-compressed raw binary data}
    - as previously saved by SaveToBinary method

function LoadFromJSON(JSON: PUTF8Char ; CustomVariantOptions:
  PDocVariantOptions=nil): boolean; overload;
    \textit{Unserialize the content from "key":value JSON object}
    - note that input JSON buffer is not modified in place: no need to create a temporary copy if the
      buffer is about to be re-used

function LoadFromJSON(const JSON: RawUTF8 ; CustomVariantOptions:
  PDocVariantOptions=nil): boolean; overload;
    \textit{Unserialize the content from "key":value JSON object}
    - if the JSON input may not be correct (i.e. if not coming from SaveToJSON), you may set
      EnsureNoKeyCollision=TRUE for a slow but safe keys validation

class function OnCanDeleteSynPersistentLock(const aKey, aValue; aIndex: integer):
    boolean;
    \textit{Can be assigned to OnCanDeleteDeprecated to check TSynPersistentLock(aValue).Safe.IsLocked}

class function OnCanDeleteSynPersistentLocked(const aKey, aValue; aIndex: integer):
    boolean;
    \textit{Can be assigned to OnCanDeleteDeprecated to check TSynPersistentLock(aValue).Safe.IsLocked}

function RawCount: integer;
    \textit{Fast returns how many items are currently stored in this dictionary}
    - this method is NOT thread-safe so should be protected by fSafe.Lock/UnLock
function SaveToBinary(NoCompression: boolean=false): RawByteString;
    Save the content as SynLZ-compressed raw binary data
    - warning: this format is tied to the values low-level RTTI, so if you change the value/key type
    definitions, LoadFromBinary() would fail

function SaveToJSON(EnumSetsAsText: boolean=false): RawUTF8; overload;
    Serialize the content as a "key":value JSON object

function SaveValuesToJSON(EnumSetsAsText: boolean=false): RawUTF8;
    Serialize the Values[] as a JSON array

function UpdateInArray(const aKey, aArrayValue): boolean;
    Replace aArrayValue item of a dynamic-array value associated via aKey
    - expect the stored value to be a dynamic array itself
    - would search for aKey as primary key, then use TDynArray.FindAndUpdate to delete any
    aArrayValue match in the associated dynamic array
    - returns FALSE if Values is not a TDynArray, or if aKey or aArrayValue were not found
    - this method is thread-safe, since it will lock the instance

procedure CopyValues(out Dest; ObjArrayByRef: boolean=false);
    Make a copy of the stored values
    - this method is thread-safe, since it will lock the instance during copy
    - resulting length(Dest) will match the exact values count
    - T*ObjArray will be reallocated and copied by content (using a temporary JSON serialization),
    unless ObjArrayByRef is true and pointers are copied

procedure DeleteAll;
    Delete all key/value stored in the current instance

procedure SaveToJSON(W: TTextWriter; EnumSetsAsText: boolean=false); overload;
    Serialize the content as a "key":value JSON object

procedure SetTimeoutAtIndex(aIndex: integer);
    Touch the entry timeout field so that it won’t be deprecated sooner
    - this method is not thread-safe, and is expected to be execute e.g. from a ForEach()
    TSynDictionaryEvent callback

property Capacity: integer read GetCapacity write SetCapacity;
    Defines how many items are currently stored in Keys/Values internal arrays

property CompressAlgo: TAlgoCompress read fCompressAlgo write fCompressAlgo;
    The compression algorithm used for binary serialization

property Keys: TDynArrayHashed read fKeys;
    Direct access to the primary key identifiers
    - if you want to access the keys, you should use fSafe.Lock/Unlock

property OnCanDeleteDeprecated: TSynDictionaryCanDeleteEvent read fOnCanDeleteDeprecated;
    Callback to by-pass DeleteDeprecated deletion by returning false
    - can be assigned e.g. to OnCanDeleteSynPersistentLock if Value is a TSynPersistentLock
    instance, to avoid any potential access violation
property TimeOut: TCardinalDynArray read fTimeOut;
  Direct low-level access to the internal access tick (GetTickCount64 shr 10)
  - may be nil if TimeOutSeconds=0

property TimeOutSeconds: cardinal read GetTimeOutSeconds;
  Returns the aTimeOutSeconds parameter value, as specified to Create()

property Values: TDynArray read fValues;
  Direct access to the associated stored values
  - if you want to access the values, you should use fSafe.Lock/Unlock

TMemoryMap = object(TObject)
  Handle memory mapping of a file content

function Map(const aFileName: TFileName): boolean; overload;
  Map the file specified by its name
  - file will be closed when UnMap will be called

function Map(aFile: THandle; aCustomSize: PtrUInt=0; aCustomOffset: Int64=0): boolean; overload;
  Map the corresponding file handle
  - if aCustomSize and aCustomOffset are specified, the corresponding map view if created (by
default, will map whole file)

procedure Map(aBuffer: pointer; aBufferSize: PtrUInt); overload;
  Set a fixed buffer for the content
  - emulated a memory-mapping from an existing buffer

procedure UnMap;
  Unmap the file

property Buffer: PAnsiChar read fBuf;
  Retrieve the memory buffer mapped to the file content

property FileHandle: THandle read fFile;
  Access to the low-level associated File handle (if any)

property FileSize: Int64 read fFileSize;
  Retrieve the mapped file size

property Size: PtrUInt read fBufSize;
  Retrieve the buffer size

TMemoryMapText = class(TObject)
  Able to read a UTF-8 text file using memory map
  - much faster than TStringList.LoadFromFile()
  - will ignore any trailing UTF-8 BOM in the file content, but will not expect one either
constructor Create(aFileContent: PUTF8Char; aFileSize: integer); overload;
    Read an UTF-8 encoded text file content
    - every line beginning is stored into LinePointers[]
    - this overloaded constructor accept an existing memory buffer (some uncompressed data e.g.)

constructor Create(const aFileName: TFileName); overload;
    Read an UTF-8 encoded text file
    - every line beginning is stored into LinePointers[]

constructor Create; overload; virtual;
    Initialize the memory mapped text file
    - this default implementation just do nothing but is called by overloaded constructors so may be
      overriden to initialize an inherited class

destructor Destroy; override;
    Release the memory map and internal LinePointers[]

function LineContains(const aUpperSearch: RawUTF8; aIndex: Integer): Boolean; virtual;
    Returns TRUE if the supplied text is contained in the corresponding line

function LineSize(aIndex: integer): integer;
    Retrieve the number of UTF-8 chars of the given line
    - warning: no range check is performed about supplied index

function LineSizeSmallerThan(aIndex, aMinimalCount: integer): boolean;
    Check if there is at least a given number of UTF-8 chars in the given line
    - this is faster than LineSize(aIndex)<aMinimalCount for big lines

procedure AddInMemoryLine(const aNewLine: RawUTF8); virtual;
    Add a new line to the already parsed content
    - this line won't be stored in the memory mapped file, but stay in memory and appended to the
      existing lines, until this instance is released

procedure AddInMemoryLinesClear; virtual;
    Clear all in-memory appended rows

procedure SaveToFile(FileName: TFileName; const Header: RawUTF8='');
    Save the whole content into a specified file
    - including any runtime appended values via AddInMemoryLine()
    - an optional header text can be added to the beginning of the file

procedure SaveToStream(Dest: TStream; const Header: RawUTF8);
    Save the whole content into a specified stream
    - including any runtime appended values via AddInMemoryLine()

property Count: integer read fCount;
    The number of text lines

property FileName: TFileName read fFileName write fFileName;
    The file name which was opened by this instance
property LinePointers: PPointerArray read fLines;
  Direct access to each text line
  - use LineSize() method to retrieve line length, since end of line will NOT end with #0, but with #13 or #10
  - warning: no range check is performed about supplied index

property Lines[aIndex: integer]: RawUTF8 read GetLine;
  Retrieve a line content as UTF-8
  - a temporary UTF-8 string is created
  - will return '' if aIndex is out of range

property Map: TMemoryMap read fMap;
  The memory map used to access the raw file content

property Strings[aIndex: integer]: string read GetString;
  Retrieve a line content as generic VCL string type
  - a temporary VCL string is created (after conversion for UNICODE Delphi)
  - will return '' if aIndex is out of range

TFakeWriterStream = class(TStream)
  A fake TStream, which will just count the number of bytes written

TRawByteStringStream = class(TStream)
  A TStream using a RawByteString as internal storage
  - default TStringStream uses WideChars since Delphi 2009, so it is not compatible with previous versions, and it does make sense to work with RawByteString in our UTF-8 oriented framework
  - just like TStringStream, is designed for appending data, not modifying in-place, as requested e.g. by TTextWriter or TFileBufferWriter classes

constructor Create(const aString: RawByteString=''); overload;
  Initialize the storage, optionally with some RawByteString content

function Read(var Buffer; Count: Longint): Longint; override;
  Read some bytes from the internal storage
  - returns the number of bytes filled into Buffer (<=Count)

function Seek(Offset: Longint; Origin: Word): Longint; override;
  Change the current Read/Write position, within current stored range

function Write(const Buffer; Count: Longint): Longint; override;
  Append some data to the buffer
  - will resize the buffer, i.e. will replace the end of the string from the current position with the supplied data

property DataString: RawByteString read fDataString write fDataString;
  Direct low-level access to the internal RawByteString storage
TSynMemoryStream = class(TCustomMemoryStream)

A TStream pointing to some in-memory data, for instance UTF-8 text
- warning: there is no local copy of the supplied content: the source data must be available during all the TSynMemoryStream usage

constructor Create(Data: pointer; DataLen: PtrInt); overload;
Create a TStream with the supplied data buffer
- warning: there is no local copy of the supplied content: the Data/DataLen buffer must be available during all the TSynMemoryStream usage: don't release the source Data before calling TSynMemoryStream.Free

constructor Create(const aText: RawByteString); overload;
Create a TStream with the supplied text data
- warning: there is no local copy of the supplied content: the aText variable must be available during all the TSynMemoryStream usage: don't release aText before calling TSynMemoryStream.Free
- aText can be on any AnsiString format, e.g. RawUTF8 or RawByteString

function Write(const Buffer; Count: Longint): Longint;
override;
This TStream is read-only: calling this method will raise an exception

TSynMemoryStreamMapped = class(TSynMemoryStream)

A TStream created from a file content, using fast memory mapping

constructor Create(aFile: THandle; aCustomSize: PtrUInt=0; aCustomOffset: Int64=0); overload;
Create a TStream from a file content using fast memory mapping
- if aCustomSize and aCustomOffset are specified, the corresponding map view if created (by default, will map whole file)

constructor Create(const aFileName: TFileName; aCustomSize: PtrUInt=0; aCustomOffset: Int64=0); overload;
Create a TStream from a file content using fast memory mapping
- if aCustomSize and aCustomOffset are specified, the corresponding map view if created (by default, will map whole file)

destructor Destroy; override;
Release any internal mapped file instance

property FileName: TFileName read fFileName;
The file name, if created from such Create(aFileName) constructor

TValuePUTF8Char = object(TObject)

Points to one value of raw UTF-8 content, decoded from a JSON buffer
- used e.g. by JSONDecode() overloaded function to returns names/values

Value: PUTF8Char;
A pointer to the actual UTF-8 text
ValueLen: PtrInt;
  How many UTF-8 bytes are stored in Value

function Idem(const Text: RawUTF8): boolean;
  Will call IdemPropNameU() over the stored text Value

function ToCardinal: PtrUInt;
  Convert the value into an unsigned integer

function ToInteger: PtrInt;
  Convert the value into a signed integer

function ToString: string;
  Convert the value into a VCL/generic string

function ToUTF8: RawUTF8; overload;
  Convert the value into a UTF-8 string

procedure ToUTF8(var Text: RawUTF8); overload;
  Convert the value into a UTF-8 string

TNameValuePUTF8Char = record
  Store one name/value pair of raw UTF-8 content, from a JSON buffer
  - used e.g. by JSONDecode() overloaded function or UrlEncodeJsonObject() to returns names/values
  Name: PUTF8Char;
    A pointer to the actual UTF-8 name text
  NameLen: integer;
    How many UTF-8 bytes are stored in Name (should be integer, not PtrInt)
  Value: PUTF8Char;
    A pointer to the actual UTF-8 value text
  ValueLen: integer;
    How many UTF-8 bytes are stored in Value

TSynLogExceptionContext = record
  Calling context of TSynLogExceptionToStr callbacks
  EAddr: PtrUInt;
    The address where the exception occurred
  EClass: ExceptClass;
    The raised exception class
  ECode: DWord;
    The OS-level exception code
    - could be $0EEDFAE0 of $0EEDFADE for Delphi-generated exceptions
EInstance: Exception;
   *The Delphi Exception instance*
   - may be nil for external/OS exceptions

ELevel: TSynLogInfo;
   *The logging level corresponding to this exception*
   - may be either sllException or sllExceptionOS

EStack: PPtrUInt;
   *The optional stack trace*

EStackCount: integer;
   = FPC's RaiseProc() FrameCount if EStack is Frame: PCodePointer

ETimestamp: TUnixTime;
   *The timestamp of this exception, as number of seconds since UNIX Epoch*
   - UnixTimeUTC is faster than NowUTC or GetSystemTime
   - use UnixTimeToDateTIme() to convert it into a regular TDateTime

ESynException = class(Exception)
   *Generic parent class of all custom Exception types of this unit*
   - all our classes inheriting from ESynException are serializable, so you could use ObjectToJSONDebug(anyESynException) to retrieve some extended information

constructor CreateLastError(const Format: RawUTF8; const Args: array of const);
   *Constructor appending some FormatUTF8() content to the GetLastError*
   - message will contain GetLastError value followed by the formatted text
   - expect % as delimiter, so is less error prone than %s %d %g
   - will handle vtPointer/vtClass/vtObject/vtVariant kind of arguments, appending class name for any class or object, the hexa value for a pointer, or the JSON representation of any supplied TDocVariant

constructor CreateUTF8(const Format: RawUTF8; const Args: array of const);
   *Constructor which will use FormatUTF8() instead of Format()*
   - expect % as delimiter, so is less error prone than %s %d %g
   - will handle vtPointer/vtClass/vtObject/vtVariant kind of arguments, appending class name for any class or object, the hexa value for a pointer, or the JSON representation of any supplied TDocVariant

function CustomLog(WR: TTextWriter; const Context: TSynLogExceptionContext): boolean; virtual;
   *Can be used to customize how the exception is logged*
   - this default implementation will call the DefaultSynLogExceptionToStr() function or the TSynLogExceptionToStrCustom global callback, if defined
   - override this method to provide a custom logging content
   - should return TRUE if Context.EAddr and Stack trace is not to be written (i.e. as for any TSynLogExceptionToStr callback)
property RaisedAt: pointer read fRaisedAt write fRaisedAt;

The code location when this exception was triggered
- populated by SynLog unit, during interception - so may be nil
- you can use TSynMapFile.FindLocation(ESynException) class function to guess the corresponding source code line
- will be serialized as "Address": hexadecimal and source code location (using TSynMapFile .map/.mab information) in TJSONSerializer.WriteObject when woStorePointer option is defined - e.g. with ObjectToJSONDebug()

EDocVariant = class(ESynException)
Exception class associated to TDocVariant JSON/BSON document

EFastReader = class(ESynException)
Exception raised during TFastReader decoding

TDWordRec = record
  Binary access to an unsigned 32-bit value (4 bytes in memory)

TQWordRec = record
  Binary access to an unsigned 64-bit value (8 bytes in memory)

THash128Rec = packed record
  Map a 128-bit hash as an array of lower bit size values
  - consumes 16 bytes of memory

THash256Rec = packed record
  Map a 256-bit hash as an array of lower bit size values
  - consumes 32 bytes of memory

THash512Rec = packed record
  Map a 512-bit hash as an array of lower bit size values
  - consumes 64 bytes of memory

TSynDate = object(TObject)
A simple way to store a date as Year/Month/Day
- with no needed computation as with TDate/TUnixTime values
- consider using TSynSystemTime if you need to handle both Date and Time
- match the first 4 fields of TSynSystemTime - so PSynDate(@aSynSystemTime)^ is safe to be used
- DayOfWeek field is not handled by its methods by default, but could be filled on demand via ComputeDayOfWeek - making this record 64-bit long

function Compare(const another: TSynDate): integer;
  Compare the stored value to a supplied value
  - returns <0 if the stored value is smaller than the supplied value, 0 if both are equals, and >0 if the stored value is bigger
  - DayOfWeek field value is not compared

function IsEqual(const another: TSynDate): boolean;
  Returns true if all fields do match - ignoring DayOfWeek field value

function IsZero: boolean;
  Returns true if all fields are zero
function ParseFromText(var P: PUTF8Char): boolean;

Try to parse a YYYY-MM-DD or YYYYMMDD ISO-8601 date from the supplied buffer
- on success, move P^ just after the date, and return TRUE

function ToDate: TDate;

Convert the stored date into a Delphi TDate floating-point value

function ToText(Expanded: boolean=true): RawUTF8;

Encode the stored date as ISO-8601 text
- returns "" if the stored date is 0 (i.e. after Clear)

procedure Clear;

Set all fields to 0

procedure ComputeDayOfWeek;

Fill the DayOfWeek field from the stored Year/Month/Day
- by default, most methods will just store 0 in the DayOfWeek field
- sunday is DayOfWeek 1, saturday is 7

procedure FromDate(date: TDate);

Fill fields with the supplied date

procedure FromNow(localtime: boolean=false);

Fill fields with the current UTC/local date, using a 8-16ms thread-safe cache

procedure SetMax;

Set internal date to 9999-12-31

TSynSystemTime = object(TObject)

A cross-platform and cross-compiler TSystemTime 128-bit structure
- FPC's TSystemTime in datih.inc does NOT match Windows TSystemTime fields!
- also used to store a Date/Time in TSynTimeZone internal structures, or for fast conversion from TDateTime to its ready-to-display members
- DayOfWeek field is not handled by most methods by default, but could be filled on demand via ComputeDayOfWeek

function EncodeForTimeChange(const aYear: word): TDateTime;

Used by TSynTimeZone

function FromText(const iso: RawUTF8): boolean;

Fill Year/Month/Day and Hour/Minute/Second fields from the given ISO-8601 text
- returns true on success

function IsDateEqual(const date: TSynDate): boolean;

Returns true if date fields do match (ignoring DayOfWeek)

function IsEqual(const another: TSynSystemTime): boolean;

Returns true if all fields do match

function IsZero: boolean;

Returns true if all fields are zero
function ToDateTime: TDateTime;
    Convert the stored time into a TDateTime

function ToNCSAText(P: PUTF8Char): PtrInt;
    Append the stored date and time, in apache-like format, to a memory buffer
    - e.g. append '19/Feb/2019:06:18:55 ' - including a trailing space
    - returns the number of chars added to P, i.e. always 21

function ToText(Expanded: boolean=true; FirstTimeChar: AnsiChar='T'; const TZD: RawUTF8=''): RawUTF8;
    Encode the stored date/time as ISO-8601 text with Milliseconds

procedure AddLogTime(WR: TTextWriter);
    Append the stored date and time, in a log-friendly format
    - e.g. append '20110325 19241502'
    - as called by TTextWriter.AddCurrentLogTime()

procedure AddNCSAText(WR: TTextWriter);
    Append the stored date and time, in apache-like format, to a TTextWriter
    - e.g. append '19/Feb/2019:06:18:55 ' - including a trailing space

procedure Clear;
    Set all fields to 0

procedure ComputeDayOfWeek;
    Fill the DayOfWeek field from the stored Year/Month/Day
    - by default, most methods will just store 0 in the DayOfWeek field
    - sunday is DayOfWeek 1, saturday is 7

procedure FromDate(const dt: TDateTime);
    Fill Year/Month/Day fields from the given value - but not DayOfWeek
    - faster than the RTL DecodeDate() function

procedure FromDateTime(const dt: TDateTime);
    Fill fields from the given value - but not DayOfWeek

procedure FromMS(ms: PtrUInt);
    Fill Hour/Minute/Second/Millisecond fields from the given number of milliseconds
    - faster than the RTL DecodeTime() function

procedure FromNowLocal;
    Fill fields with the current Local time, using a 8-16ms thread-safe cache

procedure FromNowUTC;
    Fill fields with the current UTC time, using a 8-16ms thread-safe cache

procedure FromSec(s: PtrUInt);
    Fill Hour/Minute/Second/Millisecond fields from the given number of seconds
    - faster than the RTL DecodeTime() function

procedure FromTime(const dt: TDateTime);
    Fill Hour/Minute/Second/Millisecond fields from the given TDateTime value
    - faster than the RTL DecodeTime() function
procedure IncrementMS(ms: integer);
    Add some 1..999 milliseconds to the stored time
    - not to be used for computation, but e.g. for fast AddLogTime generation

procedure ToHTTPDate(out text: RawUTF8; const tz: RawUTF8='GMT');
    Convert the stored date and time to its text in HTTP-like format
    - i.e. "Tue, 15 Nov 1994 12:45:26 GMT" to be used as a value of "Date", "Expires" or "Last-Modified" HTTP header
    - handle UTC/GMT time zone by default

procedure ToIsoDate(out text: RawUTF8);
    Convert the stored date into its Iso-8601 text with no time part

procedure ToIsoDateTime(out text: RawUTF8; const FirstTimeChar: AnsiChar='T');
    Convert the stored date and time into its Iso-8601 text, with no Milliseconds

procedure ToIsoTime(out text: RawUTF8; const FirstTimeChar: RawUTF8='T');
    Convert the stored time into its Iso-8601 text with no date part nor Milliseconds

procedure ToSynDate(out date: TSynDate);
    Copy Year/Month/DayOfWeek/Day fields to a TSynDate

TTimeLogBits = object(TObject)
    Internal memory structure for direct access to a TTimeLog type value
    - most of the time, you should not use this object, but higher level
    TimeLogFromDateTime/TimeLogToDateTime/TimeLogNow/Iso8601ToTimeLog functions
    - since TTimeLogBits.Value is bit-oriented, you can't just add or substract two TTimeLog values
    when doing date/time computation: use a TDateTime temporary conversion in such case
    - TTimeLogBits.Value needs up to 40-bit precision, so features exact representation as JavaScript
    numbers (stored in a 52-bit mantissa)

Value: Int64;
    The bit-encoded value itself, which follows an abstract "year" of 16 months of 32 days of 32
    hours of 64 minutes of 64 seconds
    - bits 0..5 = Seconds (0..59)
    - bits 6..11 = Minutes (0..59)
    - bits 12..16 = Hours (0..23)
    - bits 17..21 = Day-1 (0..31)
    - bits 22..25 = Month-1 (0..11)
    - bits 26..40 = Year (0..9999)

function Day: Integer;
    Get the day (1..31) of the TTimeLog value

function FullText(Expanded: boolean; FirstTimeChar: AnsiChar = 'T'; QuotedChar: AnsiChar = #0): RawUTF8; overload;
    Convert to Iso-8601 encoded text with date and time part
    - never truncate to date/time nor return "" as Text() does
function FullText(Dest: PUTF8Char; Expanded: boolean; FirstTimeChar: AnsiChar = 'T';
    QuotedChar: AnsiChar = #0): PUTF8Char; overload;
    URW1111 on Delphi 2010 and URW1136 on XE convert to Iso-8601 encoded text with date and
time part
    - never truncate to date/time or return " as Text() does

function Hour: integer;
    Get the hour (0..23) of the TTimeLog value

function i18nText: string;
    Convert to ready-to-be displayed text
    - using i18nDateText global event, if set (e.g. by mORMoti18n.pas)

function Minute: integer;
    Get the minute (0..59) of the TTimeLog value

function Month: Integer;
    Get the month (1..12) of the TTimeLog value

function Second: integer;
    Get the second (0..59) of the TTimeLog value

function Text(Dest: PUTF8Char; Expanded: boolean; FirstTimeChar: AnsiChar = 'T');
    integer; overload;
Convert to Iso-8601 encoded text, truncated to date/time only if needed

function Text(Expanded: boolean; FirstTimeChar: AnsiChar = 'T'): RawUTF8; overload;
Convert to Iso-8601 encoded text, truncated to date/time only if needed

function ToDate: TDateTime;
    Convert to a Delphi Date
    - will return 0 if the stored value is not a valid date

function ToDateTime: TDateTime;
    Convert to a Delphi Date and Time
    - will return 0 if the stored value is not a valid date

function ToTime: TDateTime;
    Convert to a Delphi Time

function ToUnixMSTime: TUnixMSTime;
    Convert to a millisecond-based c-encoded time (from Unix epoch 1/1/1970)
    - of course, milliseconds will be 0 due to TTimeLog second resolution

function ToUnixTime: TUnixTime;
    Convert to a second-based c-encoded time (from Unix epoch 1/1/1970)

function Year: Integer;
    Get the year (e.g. 2015) of the TTimeLog value

procedure Expand(out Date: TSynSystemTime);
    Extract the date and time content in Value into individual values

procedure From(P: PUTF8Char; L: integer); overload;
    Fill Value from Iso-8601 encoded text
procedure From(const S: RawUTF8); overload;
  Fill Value from Iso-8601 encoded text

procedure From(FileDate: integer); overload;
  Fill Value from specified File Date

procedure From(Y,M,D, HH,MM,SS: cardinal); overload;
  Fill Value from specified Date and Time

procedure From(DateTime: TDateTime; DateOnly: Boolean=false); overload;
  Fill Value from specified TDateTime

procedure From(Time: PSynSystemTime); overload;
  Fill Value from specified Date/Time individual fields

procedure FromNow;
  Fill Value from current local system Date and Time

procedure FromUnixMSTime(const UnixMSTime: TUnixMSTime);
  Fill Value from millisecond-based c-encoded time (from Unix epoch 1/1/1970)
  - of course, millisecond resolution will be lost during conversion

procedure FromUnixTime(const UnixTime: TUnixTime);
  Fill Value from second-based c-encoded time (from Unix epoch 1/1/1970)

procedure FromUTCTime;
  Fill Value from current UTC system Date and Time
  - FromNow uses local time: this function retrieves the system time expressed in Coordinated Universal Time (UTC)

TFileVersion = class(TObject)
To have existing RTTI for published properties used to retrieve version information from any EXE
- under Linux, all version numbers are set to 0 by default
- you should not have to use this class directly, but via the ExeVersion global variable

Build: Integer;
  Executable release build number

BuildYear: word;
  Build year of this exe file

Comments: RawUTF8;
  Associated Comments string version resource
  - only available on Windows - contains " under Linux/POSIX

CompanyName: RawUTF8;
  Associated CompanyName string version resource
  - only available on Windows - contains " under Linux/POSIX

FileDescription: RawUTF8;
  Associated FileDescription string version resource
  - only available on Windows - contains " under Linux/POSIX
FileVersion: RawUTF8;

Associated FileVersion string version resource
- only available on Windows - contains " under Linux/POSIX

InternalName: RawUTF8;

Associated InternalName string version resource
- only available on Windows - contains " under Linux/POSIX

LegalCopyright: RawUTF8;

Associated LegalCopyright string version resource
- only available on Windows - contains " under Linux/POSIX

Main: string;

Version info of the exe file as '3.1'
- return "string" type, i.e. UnicodeString for Delphi 2009+

Major: Integer;

Executable major version number

Minor: Integer;

Executable minor version number

OriginalFilename: RawUTF8;

Associated OriginalFileName string version resource
- only available on Windows - contains " under Linux/POSIX

ProductName: RawUTF8;

Associated ProductName string version resource
- only available on Windows - contains " under Linux/POSIX

ProductVersion: RawUTF8;

Associated ProductVersion string version resource
- only available on Windows - contains " under Linux/POSIX

Release: Integer;

Executable release version number

constructor Create(const aFileName: TFileName; aMajor: integer=0; aMinor: integer=0; aRelease: integer=0; aBuild: integer=0);

Retrieve application version from exe file name
- DefaultVersion32 is used if no information Version was included into the executable resources (on compilation time)
- you should not have to use this constructor, but rather access the ExeVersion global variable

function BuildDateTimeString: string;

Build date and time of this exe file, as plain text

function DetailedOrVoid: string;

Version info of the exe file as '3.1.0.123' or "
- this method returns " if Detailed is '0.0.0.0'
class function GetVersionInfo(const aFileName: TFileName): RawUTF8;
  Returns the version information of a specified exe file as text
  - includes FileName (without path), Detailed and BuildDateTime properties
  - e.g. 'myprogram.exe 3.1.0.123 2016-06-14 19:07:55'

function UserAgent: RawUTF8;
  Returns a ready-to-use User-Agent header with exe name, version and OS
  - e.g. 'myprogram/3.1.0.123W32' for myprogram running on Win32
  - here OS_INITIAL[] character is used to identify the OS, with '32' appended on Win32 only (e.g. 'myprogram/3.1.0.2W', is for Win64)

function Version32: integer;
  Retrieve the version as a 32-bit integer with Major.Minor.Release
  - following Major shl 16+Minor shl 8+Release bit pattern

function VersionInfo: RawUTF8;
  Returns the version information of this exe file as text
  - includes FileName (without path), Detailed and BuildDateTime properties
  - e.g. 'myprogram.exe 3.1.0.123 (2016-06-14 19:07:55)'

property BuildDateTime: TDateTime read fBuildDateTime write fBuildDateTime;
  Build date and time of this exe file

property Detailed: string read fDetailed write fDetailed;
  Version info of the exe file as '3.1.0.123'
  - return "string" type, i.e. UnicodeString for Delphi 2009+
  - under Linux, always return '0.0.0.0' if no custom version number has been defined
  - consider using DetailedOrVoid method if '0.0.0.0' is not expected

TOperatingSystemVersion = packed record
  The running Operating System, encoded as a 32-bit integer

TOSVersionInfoEx = record
  Low-level API structure, not defined in older Delphi versions

TWinRegistry = object(TObject)
  Direct access to the Windows Registry
  - could be used as alternative to TRegistry, which doesn't behave the same on all Delphi versions, and is enhanced on FPC (e.g. which supports REG_MULTI_SZ)
  - is also Unicode ready for text, using UTF-8 conversion on all compilers

key: HKEY;
  The opened HKEY handle

function ReadData(const entry: SynUnicode): RawByteString;
  Low-level read a Windows Registry content after ReadOpen()
  - works with any kind of key, but was designed for REG_BINARY

function ReadDword(const entry: SynUnicode): cardinal;
  Low-level read a Windows Registry 32-bit REG_DWORD value after ReadOpen()
function ReadEnumEntries: TRawUTF8DynArray;
  Low-level enumeration of all sub-entries names of a Windows Registry key

function ReadOpen(root: HKEY; const keyname: RawUTF8; closefirst: boolean=false): boolean;
  Start low-level read access to a Windows Registry node
  - on success (returned true), ReadClose() should be called

function ReadQword(const entry: SynUnicode): QWord;
  Low-level read a Windows Registry 64-bit REG_QWORD value after ReadOpen()

function ReadString(const entry: SynUnicode; andtrim: boolean=true): RawUTF8;
  Low-level read a string from the Windows Registry after ReadOpen()
  - in respect to Delphi’s TRegistry, will properly handle REG_MULTI_SZ (return the first value of the multi-list)

procedure Close;
  Finalize low-level read access to the Windows Registry after ReadOpen()

TExeVersion = record
  Stores some global information about the current executable and computer

  Hash: THash128Rec;
    Some hash representation of this information
    - the very same executable on the very same computer run by the very same user will always have the same Hash value
    - is computed from the crc32c of this TExeVersion fields: c0 from Version32, CpuFeatures and Host, c1 from User, c2 from ProgramFullSpec and c3 from InstanceFileName
    - may be used as an entropy seed, or to identify a process execution

  Host: RawUTF8;
    The current computer host name

  InstanceFileName: TFileName;
    The full path of the running executable or library
    - for an executable, same as paramstr(0)
    - for a library, will contain the whole .dll file name

  ProgramFileName: TFileName;
    The main executable file name (including full path)
    - same as paramstr(0)

  ProgramFilePath: TFileName;
    The main executable full path (excluding .exe file name)
    - same as ExtractFilePath(paramstr(0))

  ProgramFullSpec: RawUTF8;
    The main executable details, as used e.g. by TSynLog
    - e.g. 'C:\Dev\lib\SQLite3\exe\TestSQL3.exe 1.2.3.123 (2011-03-29 11:09:06)'

  ProgramName: RawUTF8;
    The main executable name, without any path nor extension
    - e.g. 'Test' for 'c:\path\to\Test.exe'
User: RawUTF8;
*The current computer user name*

Version: TFileVersion;
*The current executable version*

**THeapMemoryStream = class(TMemoryStream)**
*To be used instead of TMemoryStream, for speed*
- allocates memory from Delphi heap (i.e. FastMM4/SynScaleMM) and not GlobalAlloc(), as was the case for oldest versions of Delphi
- uses bigger growing size of the capacity
- consider using TRawByteStringStream, as we do in our units

**TSynInvokeableVariantType = class(TInvokeableVariantType)**
*Custom variant handler with easier/faster access of variant properties, and JSON serialization support*
- default GetProperty/SetProperty methods are called via some protected virtual IntGet/IntSet methods, with less overhead (to be overridden)
- these kind of custom variants will be faster than the default TInvokeableVariantType for properties getter/setter, but you should manually register each type by calling SynRegisterCustomVariantType()
- also feature custom JSON parsing, via TryJSONToVariant() protected method

**function FindSynVariantType(aVarType: Word; out CustomType: TSynInvokeableVariantType): boolean;**
*Search of a registered custom variant type from its low-level VarType*
- will first compare with its own VarType for efficiency

**function GetProperty(var Dest: TVarData; const V: TVarData; const Name: String): Boolean; override;**
*Retrieve the field/column value*
- this method will call protected IntGet abstract method

**function IsOfType(const V: variant): boolean;**
*Returns TRUE if the supplied variant is of the exact custom type*

**function IterateCount(const V: TVarData): integer; virtual;**
*Will check if the value is an array, and return the number of items*
- if the document is an array, will return the items count (0 meaning void array) - used e.g. by TSynMustacheContextVariant
- this default implementation will return -1 (meaning this is not an array)
- overridden method could implement it, e.g. for TDocVariant of kind dvArray

**function SetProperty(const V: TVarData; const Name: string; const Value: TVarData): Boolean; override;**
*Set the field/column value*
- this method will call protected IntSet abstract method
function TryJSONToVariant(var JSON: PUTF8Char; var Value: variant; EndOfObject: PUTF8Char): boolean; virtual;

Customization of JSON parsing into variants
- will be called by e.g. by VariantLoadJSON() or GetVariantFromJSON() with Options:
  PDocVariantOptions parameter not nil
- this default implementation will always returns FALSE, meaning that the supplied JSON is not
to be handled by this custom (abstract) variant type
- this method could be overridden to identify any custom JSON content and convert it into a
dedicated variant instance, then return TRUE
- warning: should NOT modify JSON buffer in-place, unless it returns true

procedure Clear(var V: TVarData); override;

Clear the content
- this default implementation will set VType := varEmpty
- override it if your custom type needs to manage its internal memory

procedure Copy(var Dest: TVarData; const Source: TVarData; const Indirect: Boolean); override;

Copy two variant content
- this default implementation will copy the TVarData memory
- override it if your custom type needs to manage its internal structure

procedure CopyByValue(var Dest: TVarData; const Source: TVarData); virtual;

Copy two variant content by value
- this default implementation will call the Copy() method
- override it if your custom types may use a by reference copy pattern

procedure Iterate(var Dest: TVarData; const V: TVarData; Index: integer); virtual;

Allow to loop over an array document
- Index should be in 0..IterateCount-1 range
- this default implementation will do nothing

procedure Lookup(var Dest: TVarData; const Instance: TVarData; FullName: PUTF8Char);

This method will allow to look for dotted name spaces, e.g. 'parent.child'
- should return Unassigned if the FullName does not match any value
- will identify TDocVariant storage, or resolve and call the generic
  TSynInvokeableVariantType.IntGet() method until nested value match

procedure ToJSON(W: TTextWriter; const Value: variant; Escape: TTextWriterKind); overload; virtual;

Customization of variant into JSON serialization
TDocVariant = class(TSynInvokeableVariantType)

A custom variant type used to store any JSON/BSON document-based content
- i.e. name/value pairs for objects, or an array of values (including nested documents), stored in a TDocVariantData memory structure
- you can use _Obj()/_ObjFast() _Arr()/_ArrFast() _Json()/_JsonFast() or _JsonFmt()/_JsonFastFmt() functions to create instances of such variants
- property access may be done via late-binding - with some restrictions for older versions of FPC, e.g. allowing to write:
  TDocVariant.NewFast(aVariant);
aVariant.Name := 'John';
aVariant.Age := 35;
writeln(aVariant.Name, ' is ', aVariant.Age, ' years old');
- it also supports a small set of pseudo-properties or pseudo-methods:
  aVariant._Count = DocVariantData(aVariant).Count
  aVariant._Kind = ord(DocVariantData(aVariant).Kind)
  aVariant._JSON = DocVariantData(aVariant).JSON
  aVariant._(i) = DocVariantData(aVariant).Value[i]
  aVariant.Name(i) = DocVariantData(aVariant).Name[i]
  aVariant.Add(aItem) = DocVariantData(aVariant).AddItem(aItem)
  aVariant._ := aItem = DocVariantData(aVariant).AddItem(aItem)
  aVariant.Exists(aName) = DocVariantData(aVariant).GetValueIndex(aName)>=0
  aVariant.Delete(i) = DocVariantData(aVariant).Delete(i)
  aVariant.Delete(aName) = DocVariantData(aVariant).Delete(aName)
  aVariant.NameIndex(aName) = DocVariantData(aVariant).GetValueIndex(aName)
- it features direct JSON serialization/unserialization, e.g.:
  assert(_Json('"one",2,3')._JSON='"one",2,3');
- it features direct trans-typing into a string encoded as JSON, e.g.:
  assert(_Json('"one",2,3')='"one",2,3');

**destructor** Destroy; override;
Finalize the stored information

**function** DoFunction(var Dest: TVarData; const V: TVarData; const Name: string; const Arguments: TVarDataArray): Boolean; override;
Low-level callback to access internal pseudo-methods
- mainly the _[(Index: integer): variant method to retrieve an item if the document is an array

**function** InternNames: TRawUTF8Interning;
Used by dvInternNames for string interning of all Names[] values

**function** InternValues: TRawUTF8Interning;
Used by dvInternValues for string interning of all RawUTF8 Values[]

**function** IterateCount(const V: TVarData): integer; override;
Will check if the value is an array, and return the number of items
- if the document is an array, will return the items count (0 meaning void array) - used e.g. by TSynMustacheContextVariant
- this overridden method will implement it for dvArray instance kind
**class function New(Options: TDocVariantOptions=[]): variant; overload;**

*Initialize a variant instance to store some document-based content*

- you can use this function to create a variant, which can be nested into another document, e.g.:
  ```pascal
aVariant := TDocVariant.New;
aVariant.id := 10;
```
- by default, every internal value will be copied, so access of nested properties can be slow - if you expect the data to be read-only or not propagated into another place, set `Options=[dvoValueCopiedByReference]` will increase the process speed a lot
- in practice, you should better use `_Obj()/_ObjFast()` functions or `TDocVariant.NewFast()`

**class function NewArray(const Items: TVariantDynArray; Options: TDocVariantOptions=[]): variant; overload;**

*Initialize a variant instance to store some document-based array content*

- array will be initialized with data supplied dynamic array of variants

**class function NewArray(const Items: array of const; Options: TDocVariantOptions=[]): variant; overload;**

*Initialize a variant instance to store some document-based array content*

- array will be initialized with data supplied as parameters, e.g.
  ```pascal
  aVariant := TDocVariant.NewArray(['one','2','3.0']);
  ```
  which is the same as:
  ```pascal
  TDocVariant.New(aVariant);
  TDocVariantData(aVariant).AddItem('one');
  TDocVariantData(aVariant).AddItem(2);
  TDocVariantData(aVariant).AddItem(3.0);
  ```
- by default, every internal value will be copied, so access of nested properties can be slow - if you expect the data to be read-only or not propagated into another place, set `Options=[dvoValueCopiedByReference]` will increase the process speed a lot
- in practice, you should better use the function `_Arr()` which is a wrapper around this class method
Synopse mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

class function NewJSON(const JSON: RawUTF8; Options:
TDocVariantOptions=[dvoReturnNullForUnknownProperty]): variant;

Initialize a variant instance to store some document-based object content from a supplied
(extended) JSON content
- in addition to the JSON RFC specification strict mode, this method will handle some BSON-like
extensions, e.g. unquoted field names
- a private copy of the incoming JSON buffer will be used, then it will call the
TDocVariantData.InitJSONInPlace() method
- to be used e.g. as:
var V: variant;
begin
V := TDocVariant.NewJSON('{"id":10,"doc":{"name":"John","birthyear":1972}}');
assert(V.id=10);
assert(V.doc.name='John');
assert(V.doc.birthYear=1972);
// and also some pseudo-properties:
assert(V._count=2);
assert(V.doc._kind=ord(dvObject));

- or with a JSON array:
V := TDocVariant.NewJSON('["one",2,3]');
assert(V._kind=ord(dvArray));
for i := 0 to V._count-1 do
writeln(V._(i));

- by default, every internal value will be copied, so access of nested properties can be slow - if
you expect the data to be read-only or not propagated into another place, add
dvoValueCopiedByReference in Options will increase the process speed a lot
- in practice, you should better use the function _Json()/_JsonFast() which are handy wrappers
around this class method
class function NewObject(const NameValuePairs: array of const; Options:
TDocVariantOptions=[]): variant;

Initialize a variant instance to store some document-based object content
- object will be initialized with data supplied two by two, as Name,Value pairs, e.g.
aVariant := TDocVariant.NewObject(['name','John','year',1972]);

which is the same as:
TDocVariant.New(aVariant);
TDocVariantData(aVariant).AddValue('name','John');
TDocVariantData(aVariant).AddValue('year',1972);

- by default, every internal value will be copied, so access of nested properties can be slow - if
you expect the data to be read-only or not propagated into another place, set
Options=[dvoValueCopiedByReference] will increase the process speed a lot
- in practice, you should better use the function _Obj() which is a wrapper around this class
method

SynCommons.pas unit - Rev. 1.18

Page 826 of 2552


**class function** NewUnique(const SourceDocVariant: variant; Options: TDocVariantOptions=[dvoReturnNullForUnknownProperty]): variant;

*Initialize a variant instance to store some document-based object content from a supplied existing TDocVariant instance. Use it on a value returned as varByRef (e.g., by _() pseudo-method), to ensure the returned variant will behave as a stand-alone value. For instance, the following:*

```
oSeasons := TDocVariant.NewUnique(o.Seasons);
```

-is the same as:*

```
oSeasons := o.Seasons;
_Unique(oSeasons);
```

-or even:*

```
oSeasons := _Copy(o.Seasons);
```

**procedure** Cast(var Dest: TVarData; const Source: TVarData); override;

*Handle type conversion* - only types processed by now are string/OleStr/UnicodeString/date

**procedure** CastTo(var Dest: TVarData; const Source: TVarData; const AVarType: TVarType); override;

*Handle type conversion* - only types processed by now are string/OleStr/UnicodeString/date

**procedure** Clear(var V: TVarData); override;

*Low-level callback to clear the content*

**procedure** Compare(const Left, Right: TVarData; var Relationship: TVarCompareResult); override;

*Compare two variant values* - it uses case-sensitive text comparison of the JSON representation of each variant (including TDocVariant instances)

**procedure** Copy(var Dest: TVarData; const Source: TVarData; const Indirect: Boolean); override;

*Low-level callback to copy two variant content* - such copy will by default be done by-value, for safety - if you are sure you will use the variants as read-only, you can set the dvoValueCopiedByReference Option to use faster by-reference copy

**procedure** CopyByValue(var Dest: TVarData; const Source: TVarData); override;

*Copy two variant content by value* - overridden method since instance may use a by-reference copy pattern

**class procedure** GetSingleOrDefault(const docVariantArray, default: variant; var result: variant);

*Will return the unique element of a TDocVariant array or a default* - if the value is a dvArray with one single item, it will this value - if the value is not a TDocVariant nor a dvArray with one single item, it wil return the default value
class procedure IsOfTypeOrNewFast(var aValue: variant);

Ensure a variant is a TDocVariant instance
- if aValue is not a TDocVariant, will create a new JSON_OPTIONS[true]

procedure Iterate(var Dest: TVarData; const V: TVarData; Index: integer); override;

Allow to loop over an array document
- Index should be in 0..IterateCount-1 range
- this default implementation will do handle dvArray instance kind

class procedure New(out aValue: variant; aOptions: TDocVariantOptions=[]); overload;

Initialize a variant instance to store some document-based content
- by default, every internal value will be copied, so access of nested properties can be slow - if you expect the data to be read-only or not propagated into another place, set aOptions=[dvoValueCopiedByReference] will increase the process speed a lot

class procedure NewFast(const aValues: array of PDocVariantData); overload;

Initialize several variant instances to store document-based content
- replace several calls to TDocVariantData.InitFast
- to be used e.g. as
var v1,v2,v3: TDocVariantData;
begi
  TDocVariant.NewFast([v1,v2,v3]);
...

class procedure NewFast(out aValue: variant); overload;

Initialize a variant instance to store per-reference document-based content
- same as New(aValue,JSON_OPTIONS[true]);
- to be used e.g. as
var v: variant;
begi
  TDocVariant.NewFast(v);
...

procedure ToJSON(W: TTextWriter; const Value: variant; Escape: TTextWriterKind); override;

This implementation will write the content as JSON object or array
TDocVariantData = object(TObject)

Memory structure used for TDocVariant storage of any JSON/BSON document-based content as variant
- i.e. name/value pairs for objects, or an array of values (including nested documents)
- you can use _Obj()/_ObjFast() _Arr()/_ArrFast() _Json()/_JsonFast() or _JsonFmt()/_JsonFastFmt() functions to create instances of such variants
- you can transtype such an allocated variant into TDocVariantData to access directly its internals (like Count or Values[]/Names[]):

```pascal
aVariantObject := TDocVariant.NewObject(['name','John','year',1972]);
aVariantObject := _ObjFast(['name','John','year',1972]);
with _Safe(aVariantObject)^ do
  for i := 0 to Count-1 do
    writeln(Names[i], '=' ,Values[i]); // for an object

aVariantArray := TDocVariant.NewArray(['one',2,3.0]);
aVariantArray := _JsonFast(['one',2,3.0]);
with _Safe(aVariantArray)^ do
  for i := 0 to Count-1 do
    writeln(Values[i]); // for an array
```
- use "with _Safe(...)^ do" and not "with TDocVariantData(...) do" as the former will handle internal variant redirection (varByRef), e.g. from late binding or assigned another TDocVariant
- Delphi "object" is buggy on stack -> also defined as record with methods

**function AddItem(const aValue: variant): integer;**

Add a value to this document, handled as array
- if instance's Kind is dvObject, it will raise an EDocVariant exception
- you can therefore write e.g.:
  TDocVariant.New(aVariant);
  Assert(TDocVariantData(aVariant).Kind=dvUndefined);
  TDocVariantData(aVariant).AddItem('one');
  Assert(TDocVariantData(aVariant).Kind=dvArray);
- returns the index of the corresponding newly added item

**function AddItemFromText(const aValue: RawUTF8; AllowVarDouble: boolean=false): integer;**

Add a value to this document, handled as array, from its text representation
- this function expects a UTF-8 text for the value, which would be converted to a variant number, if possible (as varInt/varInt64/varCurrency unless AllowVarDouble is set)
- if instance's Kind is dvObject, it will raise an EDocVariant exception
- returns the index of the corresponding newly added item

**function AddItemText(const aValue: RawUTF8): integer;**

Add a RawUTF8 value to this document, handled as array
- if instance's Kind is dvObject, it will raise an EDocVariant exception
- returns the index of the corresponding newly added item

**function AddOrUpdateValue(const aName: RawUTF8; const aValue: variant; wasAdded: PBoolean=nil; OnlyAddMissing: boolean=false): integer;**

Add a value in this document, or update an existing entry
- if instance's Kind is dvArray, it will raise an EDocVariant exception
- any existing Name would be updated with the new Value, unless OnlyAddMissing is set to TRUE, in which case existing values would remain
- returns the index of the corresponding value, which may be just added
function AddValue(aName: PUTF8Char; aNameLen: integer; const aValue: variant; aVarValueOwned: boolean=false): integer; overload;

Add a value in this document
- overloaded function accepting a UTF-8 encoded buffer for the name

function AddValue(const aName: RawUTF8; const aVarValue: variant; aVarValueOwned: boolean=false): integer; overload;

Add a value in this document
- if aName is set, if dvCheckForDuplicatedNames option is set, any existing duplicated aName will raise an EDocVariant; if instance's kind is dvArray and aName is defined, it will raise an EDocVariant
- aName may be " e.g. if you want to store an array: in this case, dvCheckForDuplicatedNames option should not be set; if instance's Kind is dvObject, it will raise an EDocVariant exception
- if aVarValueOwned is true, then the supplied aVarValue will be assigned to the internal values - by default, it will use SetVARIANTByValue()
- you can therefore write e.g.:  
  TDocVariant.New(aVariant);  
  Assert(TDocVariantData(aVariant).Kind=dvUndefined);  
  TDocVariantData(aVariant).AddValue('name','John');  
  Assert(TDocVariantData(aVariant).Kind=dvObject);
- returns the index of the corresponding newly added value

function AddValueFromText(const aName, aVarValue: RawUTF8; Update: boolean=false; AllowVarDouble: boolean=false): integer;

Add a value in this document, from its text representation
- this function expects a UTF-8 text for the value, which would be converted to a variant number, if possible (as varInt/varInt64/varCurrency and/or as varDouble is AllowVarDouble is set)
- if Update=TRUE, will set the property, even if it is existing

function Delete(Index: integer): boolean; overload;

Delete a value/item in this document, from its index
- return TRUE on success, FALSE if the supplied index is not correct

function Delete(const aName: RawUTF8): boolean; overload;

Delete a value/item in this document, from its name
- return TRUE on success, FALSE if the supplied name does not exist

function DeleteByProp(const aPropName, aVarValue: RawUTF8; aPropValueCaseSensitive: boolean): boolean;

Delete a value in this document, by property name match
- {aPropName:aVarValue} will be searched within the stored array or object, and the corresponding item will be deleted, on match
- returns FALSE if no match is found, TRUE if found and deleted
- will call VariantEquals() for value comparison

function DeleteByStartName(aStartName: PUTF8Char; aStartNameLen: integer): integer;

Delete all values matching the first characters of a property name
- returns the number of deleted items
- returns 0 if the document is not a dvObject, or if no match was found
- will use IdemPChar(), so search would be case-insensitive
function DeleteByValue(const aValue: Variant; CaseInsensitive: boolean=false): integer;

Delete one or several value/item in this document, from its value
- returns the number of deleted items
- returns 0 if the document is not a dvObject, or if no match was found
- if the value exists several times, all occurrences would be removed
- is optimized for DeleteByValue(null) call

function FlattenAsNestedObject(const aObjectPropName: RawUTF8): boolean;

Map {"obj.prop1"..,"obj.prop2"..} into {"obj":{"prop1"..,"prop2"..}}
- the supplied aObjectPropName should match the incoming dotted value of all properties (e.g. 'obj' for "obj.prop1")
- if any of the incoming property is not of "obj.prop#" form, the whole process would be ignored
- return FALSE if the TDocVariant did not change
- return TRUE if the TDocVariant has been flattened

function GetAsBoolean(const aName: RawUTF8; out aValue: boolean; aSortedCompare: TUTF8Compare=nil): Boolean;

Find an item in this document, and returns its value as boolean
- return false if aName is not found, or if the instance is not a TDocVariant
- return true if the name has been found, and aValue stores the value
- after a SortByName(aSortedCompare), could use faster binary search
- consider using B[] property if you want simple read/write typed access

function GetAsDocVariant(const aName: RawUTF8; out aValue: PDocVariantData; aSortedCompare: TUTF8Compare=nil): boolean; overload;

Find an item in this document, and returns its value as a TDocVariantData
- return false if aName is not found, or if the instance is not a TDocVariant
- return true if the name has been found and points to a TDocVariant: then aValue stores a pointer to the value
- after a SortByName(aSortedCompare), could use faster binary search

function GetAsDocVariantSafe(const aName: RawUTF8; aSortedCompare: TUTF8Compare=nil): PDocVariantData;

Find an item in this document, and returns its value as a TDocVariantData
- returns a void TDocVariant if aName is not a document
- after a SortByName(aSortedCompare), could use faster binary search
- consider using O[] or A[] properties if you want simple read-only access, or O_[] or A_[] properties if you want the ability to add a missing object or array in the document

function GetAsDouble(const aName: RawUTF8; out aValue: double; aSortedCompare: TUTF8Compare=nil): Boolean;

Find an item in this document, and returns its value as floating point
- return false if aName is not found, or if the instance is not a TDocVariant
- return true if the name has been found, and aValue stores the value
- after a SortByName(aSortedCompare), could use faster binary search
- consider using D[] property if you want simple read/write typed access
function GetAsInt64(const aName: RawUTF8; var aValue: Int64; aSortedCompare: TUTF8Compare=nil): Boolean;

*Find an item in this document, and returns its value as integer*
- return false if aName is not found, or if the instance is not a TDocVariant
- return true if the name has been found, and aValue stores the value
- after a SortByName(aSortedCompare), could use faster binary search
- consider using [] property if you want simple read/write typed access

function GetAsInteger(const aName: RawUTF8; var aValue: integer; aSortedCompare: TUTF8Compare=nil): Boolean;

*Find an item in this document, and returns its value as integer*
- return false if aName is not found, or if the instance is not a TDocVariant
- return true if the name has been found, and aValue stores the value
- after a SortByName(aSortedCompare), could use faster binary search
- consider using [] property if you want simple read/write typed access

function GetAsPVariant(aName: PUTF8Char; aNameLen:.PtrInt): PVariant; overload;

*Find an item in this document, and returns pointer to its value*
- lookup the value by aName/aNameLen for an object document, or accept an integer text as index for an array document
- return nil if aName is not found, or if the instance is not a TDocVariant
- return a pointer to the stored variant, if the name has been found

function GetAsPVariant(const aName: RawUTF8; var aValue: PVariant; aSortedCompare: TUTF8Compare=nil): boolean; overload;

*Find an item in this document, and returns pointer to its value*
- return false if aName is not found
- return true if the name has been found: then aValue stores a pointer to the value
- after a SortByName(aSortedCompare), could use faster binary search

function GetAsRawUTF8(const aName: RawUTF8; var aValue: RawUTF8; aSortedCompare: TUTF8Compare=nil): Boolean;

*Find an item in this document, and returns its value as RawUTF8*
- return false if aName is not found, or if the instance is not a TDocVariant
- return true if the name has been found, and aValue stores the value
- after a SortByName(aSortedCompare), could use faster binary search
- consider using U[] property if you want simple read/write typed access

function GetDocVariantByPath(const aPath: RawUTF8; var aValue: PDocVariantData): boolean;

*Retrieve a reference to a TDocVariant, given its path*
- path is defined as a dotted name-space, e.g. 'doc.glossary.title'
- if the supplied aPath does not match any object, it will return false
- if aPath stores a valid TDocVariant, returns true and a pointer to it

function GetDocVariantByProp(const aPropName,aPropValue: RawUTF8; aPropValueCaseSensitive: boolean; var Dest: PDocVariantData): boolean;

*Retrieve a reference to a dvObject in the dvArray, from a property value*
- [aPropName:aPropValue] will be searched within the stored array, and the corresponding item will be copied into Dest, on match
- returns FALSE if no match is found, TRUE if found and copied by reference
function GetItemByProp(const aPropName,aPropValue: RawUTF8; aPropValueCaseSensitive: boolean; var Dest: variant; DestByRef: boolean=false): boolean;

Retrieve a dvObject in the dvArray, from a property value
- [aPropName:aPropValue] will be searched within the stored array, and the corresponding item will be copied into Dest, on match
- returns FALSE if no match is found, TRUE if found and copied
- create a copy of the variant by default, unless DestByRef is TRUE
- will call VariantEquals() for value comparison

function GetJsonByStartName(const aStartName: RawUTF8): RawUTF8;

Returns a JSON object containing all properties matching the first characters of the supplied property name
- returns null if the document is not a dvObject
- will use IdemPChar(), so search would be case-insensitive

function GetPVariantByPath(const aPath: RawUTF8): PVariant;

Retrieve a reference to a value, given its path
- path is defined as a dotted name-space, e.g. 'doc.glossary.title'
- if the supplied aPath does not match any object, it will return nil
- if aPath is found, returns a pointer to the corresponding value

function GetValueByPath(const aPath: RawUTF8): variant; overload;

Retrieve a value, given its path
- path is defined as a dotted name-space, e.g. 'doc.glossary.title'
- it will return Unassigned if the path does match the supplied aPath

function GetValueByPath(const aPath: RawUTF8; out aValue: variant): boolean; overload;

Retrieve a value, given its path
- path is defined as a dotted name-space, e.g. 'doc.glossary.title'
- it will return FALSE if the path does not match the supplied aPath
- returns TRUE and set the found value in aValue

function GetValueByPath(const aDocVariantPath: array of RawUTF8): variant; overload;

Retrieve a value, given its path
- path is defined as a list of names, e.g. ['doc','glossary','title']
- it will return Unassigned if the path does not match the data
- this method will only handle nested TDocVariant values: use the slightly slower GetValueByPath() overloaded method, if any nested object may be of another type (e.g. a TBSONVariant)

function GetValueEnumerate(const aName: RawUTF8; aTypeInfo: pointer; out aValue; aDeleteFoundEntry: boolean=false): Boolean;

Find an item in this document, and returns its value as enumerate
- return false if aName is not found, if the instance is not a TDocVariant, or if the value is not a string corresponding to the supplied enumerate
- return true if the name has been found, and aValue stores the value
- will call Delete() on the found entry, if aDeleteFoundEntry is true
function GetValueIndex(aName: PUTF8Char; aNameLen: PtrInt; aCaseSensitive: boolean): integer; overload;

Find an item index in this document from its name
- lookup the value by name for an object document, or accept an integer text as index for an array document
- returns -1 if not found

function GetValueIndex(const aName: RawUTF8): integer; overload;

Find an item index in this document from its name
- search will follow dvoNameCaseSensitive option of this document
- lookup the value by name for an object document, or accept an integer text as index for an array document
- returns -1 if not found

function GetValueOrDefault(const aName: RawUTF8; const aDefault: variant): variant;

Find an item in this document, and returns its value
- return the supplied default if aName is not found, or if the instance is not a TDocVariant

function GetValueOrEmpty(const aName: RawUTF8): variant;

Find an item in this document, and returns its value
- return a cleared variant if aName is not found, or if the instance is not a TDocVariant

function GetValueOrNull(const aName: RawUTF8): variant;

Find an item in this document, and returns its value
- return null if aName is not found, or if the instance is not a TDocVariant

function GetValueOrRaiseException(const aName: RawUTF8): variant;

Find an item in this document, and returns its value
- raise an EDocVariant if not found and dvoReturnNullForUnknownProperty is not set in Options (in this case, it will return Null)

function GetValuesByStartName(const aStartName: RawUTF8; TrimLeftStartName: boolean=false): variant;

Returns a TDocVariant object containing all properties matching the first characters of the supplied property name
- returns null if the document is not a dvObject
- will use IdemPChar(), so search would be case-insensitive

function GetVarData(const aName: RawUTF8; var aValue: TVarData; aSortedCompare: TUTF8Compare=nil): boolean; overload;

Find an item in this document, and returns its value as TVarData
- return false if aName is not found, or if the instance is not a TDocVariant
- return true and set aValue if the name has been found
- will use simple loop lookup to identify the name, unless aSortedCompare is set, and would let use a faster O(log(n)) binary search after a SortByName()

function GetVarData(const aName: RawUTF8; aSortedCompare: TUTF8Compare=nil): PVarData; overload;

Find an item in this document, and returns its value as TVarData pointer
- return nil if aName is not found, or if the instance is not a TDocVariant
- return a pointer to the value if the name has been found
- after a SortByName(aSortedCompare), could use faster binary search
function InitJSON(const JSON: RawUTF8; aOptions: TDocVariantOptions=[]): boolean;

Initialize a variant instance to store some document-based object content from a supplied JSON array of JSON object content
- a private copy of the incoming JSON buffer will be used, then it will call the other overloaded InitJSONInPlace() method
- this method is called e.g. by _Json() and _JsonFast() global functions
- if you call Init*() methods in a row, ensure you call Clear in-between

function InitJSONFromFile(const JsonFile: TFileName; aOptions: TDocVariantOptions=[]; RemoveComments: boolean=false): boolean;

Initialize a variant instance to store some document-based object content from a JSON array of JSON object content, stored in a file
- any kind of file encoding will be handled, via AnyTextFileToRawUTF8()
- you can optionally remove any comment from the file content
- if you call Init*() methods in a row, ensure you call Clear in-between

function InitJSONInPlace(JSON: PUTF8Char; aOptions: TDocVariantOptions=[]; aEndOfObject: PUTF8Char=nil): PUTF8Char;

Initialize a variant instance to store some document-based object content from a supplied JSON array or JSON object content
- warning: the incoming JSON buffer will be modified in-place: so you should make a private copy before running this method, e.g. using TSynTempBuffer
- this method is called e.g. by _JsonFmt() _JsonFastFmt() global functions with a temporary JSON buffer content created from a set of parameters
- if you call Init*() methods in a row, ensure you call Clear in-between

function InternalAdd(const aName: RawUTF8): integer;

Low-level method called internally to reserve place for new values
- returns the index of the newly created item in Values[]/Names[] arrays
- you should not have to use it, unless you want to add some items directly within the Values[]/Names[] arrays, using e.g. InitFast(InitialCapacity) to initialize the document
- if aName='', append a dvArray item, otherwise append a dvObject field
- warning: FPC optimizer is confused by Values[InternalAdd(name)] so you should call InternalAdd() in an explicit previous step

function Reduce(const aPropNames: array of RawUTF8; aCaseSensitive: boolean; aDoNotAddVoidProp: boolean=false): variant; overload;

Create a TDocVariant object, from a selection of properties of this document, by property name
- always returns a TDocVariantData, even if no property name did match (in this case, it is dvUndefined)

function ReduceAsArray(const aPropName: RawUTF8; OnReduce: TOnReducePerItem=nil): variant; overload;

Create a TDocVariant array, from the values of a single properties of this document, specified by name
- always returns a TDocVariantData, even if no property name did match (in this case, it is dvUndefined)
- you can optionally apply an additional filter to each reduced item
### function ReduceAsArray(const aPropName: RawUTF8; OnReduce: TOnReducePerValue):

Create a TDocVariant array, from the values of a single properties of this document, specified by name
- always returns a TDocVariantData, even if no property name did match (in this case, it is dvUndefined)
- this overloaded method accepts an additional filter to each reduced item

```pascal
function ReduceAsArray(const aPropName: RawUTF8; OnReduce: TOnReducePerValue): variant; overload;
```

### function Rename(const aFromPropName, aToPropName: TRawUTF8DynArray):

Rename some properties of a TDocVariant object
- returns the number of property names modified

```pascal
function Rename(const aFromPropName, aToPropName: TRawUTF8DynArray): integer;
```

### function RetrieveValueOrRaiseException(aName: PUTF8Char; aNameLen: integer; aCaseSensitive: boolean; var Dest: variant; DestByRef: boolean): boolean; overload;

Find an item in this document, and returns its value
- raise an EDocVariant if not found and dvoReturnNullForUnknownProperty is not set in Options (in this case, it will return Null)
- create a copy of the variant by default, unless DestByRef is TRUE

```pascal
function RetrieveValueOrRaiseException(aName: PUTF8Char; aNameLen: integer; aCaseSensitive: boolean; var Dest: variant; DestByRef: boolean): boolean; overload;
```

### function SearchItemByProp(const aPropNameFmt: RawUTF8; const aPropNameArgs: array of const; const aPropValue: RawUTF8; aPropValueCaseSensitive: boolean):

Search a property match in this document, handled as array or object
- {aPropName:aPropValue} will be searched within the stored array or object, and the corresponding item index will be returned, on match
- returns -1 if no match is found
- will call VariantEquals() for value comparison

```pascal
function SearchItemByProp(const aPropNameFmt: RawUTF8; const aPropNameArgs: array of const; const aPropValue: RawUTF8; aPropValueCaseSensitive: boolean): integer; overload;
```

### function SearchItemByValue(const aValue: Variant; CaseInsensitive: boolean=false; StartIndex: integer=0):

Search a value in this document, handled as array
- aValue will be searched within the stored array and the corresponding item index will be returned, on match
- returns -1 if no match is found
- you could make several searches, using the StartIndex optional parameter

```pascal
function SearchItemByValue(const aValue: Variant; CaseInsensitive: boolean=false; StartIndex: integer=0): integer;
```
function ToArrayOfConst: TTVarRecDynArray; overload;

Save an array document as an array of TVarRec, i.e. an array of const
- will expect the document to be a dvArray - otherwise, will raise a EDocVariant exception
- would allow to write code as such:
  Doc.InitArray(['one',2,3]);
  s := FormatUTF8('[%,%,%]',Doc.ToArrayOfConst,[],true);
  // here s='[one,2,3]') since % would be replaced by Args[] parameters
s := FormatUTF8('[?,?,?]',[''],Doc.ToArrayOfConst,true);
  // here s='["one",2,3]') since ? would be escaped by Params[] parameters

function ToCSV(const Separator: RawUTF8=','): RawUTF8;

Save a document as an CSV of UTF-8 encoded JSON
- will expect the document to be a dvArray - otherwise, will raise a EDocVariant exception
- will use VariantToUTF8() to populate the result array: as a consequence, any nested custom
  variant types (e.g. TDocVariant) will be stored as JSON

function ToJSON(const Prefix: RawUTF8=''; const Suffix: RawUTF8=''; Format: TTextWriterJSONFormat=jsonCompact): RawUTF8;

Save a document as UTF-8 encoded JSON
- will write either a JSON object or array, depending of the internal layout of this instance (i.e.
  Kind property value)
- will write 'null' if Kind is dvUndefined
- implemented as just a wrapper around VariantSaveJSON()

function ToNonExpandedJSON: RawUTF8;

Save an array of objects as UTF-8 encoded non expanded layout JSON
- returned content would be a JSON object in mORMot's TSQLTable non expanded format, with
  reduced JSON size, i.e.
  {"fieldCount":3,"values":{"ID","FirstName","LastName","..."}}
- will write " if Kind is dvUndefined or dvObject
- will raise an exception if the array document is not an array of objects with identical field
  names

function ToRawUTF8DynArray: TRawUTF8DynArray; overload;

Save a document as an array of UTF-8 encoded JSON
- will expect the document to be a dvArray - otherwise, will raise a EDocVariant exception
- will use VariantToUTF8() to populate the result array: as a consequence, any nested custom
  variant types (e.g. TDocVariant) will be stored as JSON

function ToTextPairs(const NameValueSep: RawUTF8='='; const ItemSep: RawUTF8=#13#10; Escape: TTextWriterKind=twJSONEscape): RawUTF8;

Save a document as UTF-8 encoded Name=Value pairs
- will follow by default the .INI format, but you can specify your own expected layout

function ToUrlEncode(const UriRoot: RawUTF8): RawUTF8;

Save an object document as an URI-encoded list of parameters
- object field names should be plain ASCII-7 RFC compatible identifiers (0..9a..zA..Z_.~),
  otherwise their values are skipped
procedure AddByPath(const aSource: TDocVariantData; const aPaths: array of RawUTF8);

Add one or several properties, specified by path, from another object
- path are defined as a dotted name-space, e.g. ‘doc.glossary.title’
- matching values would be added as root values, with the path as name
- instance and supplied aSource should be a dvObject

procedure AddFrom(const aDocVariant: Variant);

Add one or several values from another document
- supplied document should be of the same kind than the current one, otherwise nothing is added

procedure AddItems(const aValue: array of const);

Add one or several values to this document, handled as array
- if instance's Kind is dvObject, it will raise an EDocVariant exception

procedure AddNameValuesToObject(const NameValuePairs: array of const);

Add some properties to a TDocVariantData dvObject
- data is supplied two by two, as Name,Value pairs
- caller should ensure that Kind=dvObject, otherwise it won't do anything
- any existing Name would be duplicated

procedure AddOrUpdateFrom(const aDocVariant: Variant; aOnlyAddMissing: boolean=false);

Add or update or on several values from another object
- current document should be an object

procedure AddOrUpdateNameValuesToObject(const NameValuePairs: array of const);

Merge some properties to a TDocVariantData dvObject
- data is supplied two by two, as Name,Value pairs
- caller should ensure that Kind=dvObject, otherwise it won't do anything
- any existing Name would be updated with the new Value

procedure AddOrUpdateObject(const NewValues: variant; OnlyAddMissing: boolean=false; RecursiveUpdate: boolean=false);

Merge some TDocVariantData dvObject properties to a TDocVariantData dvObject
- data is supplied two by two, as Name,Value pairs
- caller should ensure that both variants have Kind=dvObject, otherwise it won't do anything
- any existing Name would be updated with the new Value, unless OnlyAddMissing is set to TRUE, in which case existing values would remain

procedure Clear;

To be called before any Init*() method call, when a previous Init*() has already be performed on the same instance, to avoid memory leaks
- for instance:

```pascal
var Doc: TDocVariantData; // stack-allocated variable
begin
  Doc.InitArray(['one',2,3.0]); // no need of any Doc.Clear here
  assert(Doc.Count=3);
  Doc.Clear; // to release memory before following InitObject()
  Doc.InitObject(['name','John','year',1972]);
end;
```

- implemented as just a wrapper around DocVariantType.Clear()
procedure FillZero;

Fill all Values[] with #0, then delete all values
- could be used to specifically remove sensitive information from memory

procedure Init(aOptions: TDocVariantOptions=[]; aKind: TDocVariantKind=dvUndefined);

Initialize a TDocVariantData to store some document-based content
- can be used with a stack-allocated TDocVariantData variable:
var Doc: TDocVariantData; // stack-allocated variable
begin
  Doc.Init;
  Doc.AddValue('name', 'John');
  assert(Doc.Value['name'] = 'John');
  assert(variant(Doc).name = 'John');
end;
- if you call Init*() methods in a row, ensure you call Clear in-between

procedure InitArray(const Items: array of const; aOptions: TDocVariantOptions=[]);

Initialize a variant instance to store some document-based array content
- array will be initialized with data supplied as parameters, e.g.
var Doc: TDocVariantData; // stack-allocated variable
begin
  Doc.InitArray(['one', 2, 3.0]);
  assert(Doc.Count = 3);
end;
which is the same as:
var Doc: TDocVariantData;
  i: integer;
begin
  Doc.Init;
  Doc.AddItem('one');
  Doc.AddItem(2);
  Doc.AddItem(3.0);
  assert(Doc.Count = 3);
  for i := 0 to Doc.Count-1 do
    writeln(Doc.Value[i]);
end;
- this method is called e.g. by _Arr() and _ArrFast() global functions
- if you call Init*() methods in a row, ensure you call Clear in-between

procedure InitArrayFrom(const Items: TRawUTF8DynArray; aOptions: TDocVariantOptions); overload;

Initialize a variant instance to store some RawUTF8 array content

procedure InitArrayFrom(const Items: TIntegerDynArray; aOptions: TDocVariantOptions); overload;

Initialize a variant instance to store some 32-bit integer array content

procedure InitArrayFrom(const Items: TInt64DynArray; aOptions: TDocVariantOptions); overload;

Initialize a variant instance to store some 64-bit integer array content

procedure InitArrayFromObjArray(const ObjArray; aOptions: TDocVariantOptions; aWriterOptions: TTextWriterWriteObjectOptions=[woDontStoreDefault]);

Initialize a variant instance to store a T*ObjArray content
- will call internally ObjectToVariant() to make the conversion
procedure InitArrayFromVariants(const Items: TVariantDynArray; aOptions: TDocVariantOptions=[]; ItemsCopiedByReference: boolean=true);

Initialize a variant instance to store some document-based array content
- array will be initialized with data supplied as variant dynamic array
- if Items is [], the variant will be set as null
- will be almost immediate, since TVariantDynArray is reference-counted, unless ItemsCopiedByReference is set to FALSE
- if you call Init*() methods in a row, ensure you call Clear in-between

procedure InitCopy(const SourceDocVariant: variant; aOptions: TDocVariantOptions);

Ensure a document-based variant instance will have one unique options set
- this will create a copy of the supplied TDocVariant instance, forcing all nested events to have the same set of Options
- you can use this function to ensure that all internal properties of this variant will be copied e.g. per-reference (if you set JSON_OPTIONS[false]) or per-value (if you set JSON_OPTIONS[false]) whatever options the nested objects or arrays were created with
- will raise an EDocVariant if the supplied variant is not a TDocVariant
- you may rather use _Unique() or _UniqueFast() wrappers if you want to ensure that a TDocVariant instance is unique
- if you call Init*() methods in a row, ensure you call Clear in-between

procedure InitCSV(CSV: PUTF8Char; aOptions: TDocVariantOptions; NameValueSep: AnsiChar='='; ItemSep: AnsiChar=#10; DoTrim: boolean=true); overload;

Initialize a variant instance to store some document-based object content from a supplied CSV UTF-8 encoded text
- the supplied content may have been generated by ToTextPairs() method
- if ItemSep=#10, then any kind of line feed (CRLF or LF) will be handled
- if you call Init*() methods in a row, ensure you call Clear in-between

procedure InitCSV(const CSV: RawUTF8; aOptions: TDocVariantOptions; NameValueSep: AnsiChar='='; ItemSep: AnsiChar=#10; DoTrim: boolean=true); overload;

Initialize a variant instance to store some document-based object content from a supplied CSV UTF-8 encoded text
- the supplied content may have been generated by ToTextPairs() method
- if ItemSep=#10, then any kind of line feed (CRLF or LF) will be handled
- if you call Init*() methods in a row, ensure you call Clear in-between

procedure InitFast; overload;

Initialize a TDocVariantData to store per-reference document-based content
- same as Doc.Init(JSON_OPTIONS[true]);
- can be used with a stack-allocated TDocVariantData variable:
var Doc: TDocVariantData; // stack-allocated variable
begin
  Doc.InitFast;
  Doc.AddValue('name', 'John');
  assert(Doc.Value['name']='John');
  assert(variant(Doc).name='John');
end;
- see also TDocVariant.NewFast() if you want to initialize several TDocVariantData variable instances at once
- if you call Init*() methods in a row, ensure you call Clear in-between
procedure InitFast(InitialCapacity: integer; aKind: TDocVariantKind); overload;

Initialize a TDocVariantData to store per-reference document-based content
- this overloaded method allows to specify an estimation of how many properties or items this
aKind document would contain

procedure InitFromTypeInfo(const aValue; aTypeInfo: pointer; aEnumSetsAsText: boolean; aOptions: TDocVariantOptions);

Initialize a variant instance to store document-based array content
- array will be initialized from the supplied variable (which would be e.g. a T*ObjArray or a
dynamic array), using RTTI
- will use a temporary JSON serialization via SaveJSON()

procedure InitObject(const NameValuePairs: array of const; aOptions: TDocVariantOptions=[]);

Initialize a TDocVariantData to store document-based object content
- object will be initialized with data supplied two by two, as Name,Value pairs, e.g.
  var Doc: TDocVariantData; // stack-allocated variable
  begin
    Doc.InitObject(['name', 'John', 'year', 1972]);
    which is the same as:
    var Doc: TDocVariantData;
    begin
      Doc.Init;
      Doc.AddValue('name', 'John');
      Doc.AddValue('year', 1972);
  - this method is called e.g. by _Obj() and _ObjFast() global functions
  - if you call Init*() methods in a row, ensure you call Clear in-between

procedure InitObjectFromPath(const aPath: RawUTF8; const aValue: variant; aOptions: TDocVariantOptions=[]);

Initialize a variant instance to store a document-based object with a single property
- the supplied path could be 'Main.Second.Third', to create nested objects, e.g.
  {"Main":{"Second":"Third":value}}
- if you call Init*() methods in a row, ensure you call Clear in-between

procedure InitObjectFromVariants(const aNames: TRawUTF8DynArray; const aValues: TVariantDynArray; aOptions: TDocVariantOptions=[]);

Initialize a variant instance to store some document-based object content
- object will be initialized with names and values supplied as dynamic arrays
  - if aNames and aValues are [] or do have matching sizes, the variant will be set as null
  - will be almost immediate, since Names and Values are reference-counted
  - if you call Init*() methods in a row, ensure you call Clear in-between

procedure Reduce(const aPropNames: array of RawUTF8; aCaseSensitive: boolean; out result: TDocVariantData; aDoNotAddVoidProp: boolean=false); overload;

Create a TDocVariant object, from a selection of properties of this document, by property name
- if the document is a dvObject, to reduction will be applied to all its properties
- if the document is a dvArray, the reduction will be applied to each stored item, if it is a document
procedure ReduceAsArray(const aPropName: RawUTF8; out result: TDocVariantData; OnReduce: TOnReducePerItem=nil); overload;

Create a TDocVariant array, from the values of a single properties of this document, specified by name
- you can optionally apply an additional filter to each reduced item

procedure ReduceAsArray(const aPropName: RawUTF8; out result: TDocVariantData; OnReduce: TOnReducePerValue); overload;

Create a TDocVariant array, from the values of a single properties of this document, specified by name
- this overloaded method accepts an additional filter to each reduced item

procedure Reset;
Delete all internal stored values
- like Clear + Init() with the same options
- will reset Kind to dvUndefined

procedure RetrieveNameOrRaiseException(Index: integer; var Dest: RawUTF8);
Retrieve an item in this document from its index, and returns its Name
- raise an EDocVariant if the supplied Index is not in the 0..Count-1 range and dvoReturnNullForUnknownProperty is set in Options

procedure RetrieveValueOrRaiseException(Index: integer; var Dest: variant; DestByRef: boolean); overload;
Retrieve an item in this document from its index, and returns its value
- raise an EDocVariant if the supplied Index is not in the 0..Count-1 range and dvoReturnNullForUnknownProperty is set in Options
- create a copy of the variant by default, unless DestByRef is TRUE

procedure Reverse;
Reverse the order of the document object or array items

procedure SetCount(aCount: integer);
Low-level method to force a number of items
- could be used to fast add items to the internal Values[]/Names[] arrays
- just set protected VCount field, do not resize the arrays: caller should ensure that Capacity is big enough

procedure SetValueOrRaiseException(Index: integer; const NewValue: variant);
Set an item in this document from its index
- raise an EDocVariant if the supplied Index is not in 0..Count-1 range

procedure SortArrayByField(const aItemPropName: RawUTF8; aValueCompare: TVariantCompare=nil; aValueCompareReverse: boolean=false; aNameSortedCompare: TUTF8Compare=nil);
Sort the document array values by a field of some stored objet values
- do nothing if the document is not a dvArray, or if the items are no dvObject
- will sort by UTF-8 text (VariantCompare) if no custom aValueCompare is supplied
procedure SortByName(Compare: TUTF8Compare=nil);

Sort the document object values by name
- do nothing if the document is not a dvObject
- will follow case-insensitive order (@StrIComp) by default, but you can specify @StrComp as comparer function for case-sensitive ordering
- once sorted, you can use GetVarData(...,Compare) or GetAs*(...,.Compare) methods for much faster O(log(n)) binary search

procedure SortByValue(Compare: TVariantCompare=nil);

Sort the document object values by value
- work for both dvObject and dvArray documents
- will sort by UTF-8 text (VariantCompare) if no custom aCompare is supplied

procedure ToArrayOfConst(out Result: TTVarRecDynArray); overload;

Save an array document as an array of TVarRec, i.e. an array of const
- will expect the document to be a dvArray - otherwise, will raise a EDocVariant exception
- would allow to write code as such:
  Doc.InitArray(['one',2,3]);
  Doc.ToArrayOfConst(vr);
  s := FormatUTF8('%[%,%,%]',vr,[],true);
  // here s='[one,2,3]') since % would be replaced by Args[] parameters
  s := FormatUTF8('%[?,?,?]',[],vr,true);
  // here s='"one",2,3]') since ? would be escaped by Params[] parameters

procedure ToRawUTF8DynArray(out Result: TRawUTF8DynArray); overload;

Save a document as an array of UTF-8 encoded JSON
- will expect the document to be a dvArray - otherwise, will raise a EDocVariant exception
- will use VariantToUTF8() to populate the result array: as a consequence, any nested custom variant types (e.g. TDocVariant) will be stored as JSON

procedure ToTextPairsVar(out result: RawUTF8; const NameValueSep: RawUTF8='=';
const ItemSep: RawUTF8=#13#10; Escape: TTextWriterKind=twJSONEscape);

Save a document as UTF-8 encoded Name=Value pairs
- will follow by default the .INI format, but you can specify your own expected layout

property A[const aName: RawUTF8]: PDocVariantData read GetArrayExistingByName;

Direct access to a dvObject existing dvArray property from its name
- follows dvoNameCaseSensitive and dvoReturnNullForUnknownProperty options
- A['prop'] would return a fake void TDocVariant if the property is not existing or not a dvArray, just like GetAsDocVariantSafe()
- use A_['prop'] to force adding any missing property

property A[const aName: RawUTF8]: PDocVariantData read GetArrayOrAddByName;

Direct access or add a dvObject's dvArray property from its name
- follows dvoNameCaseSensitive and dvoReturnNullForUnknownProperty options
- A_['prop'] would add a new property if there is none existing, or overwrite an existing property which is not a dvArray
property B[const aName: RawUTF8]: Boolean read GetBooleanByName write SetBooleanByName;

Direct access to a dvObject Boolean stored property value from its name
- slightly faster than the variant-based Value[] default property
- follows dvoNameCaseSensitive and dvoReturnNullForUnknownProperty options
- use GetAsBoolean if you want to check the availability of the field
- B['prop'] := true would add a new property, or overwrite an existing

property Capacity: integer read GetCapacity write SetCapacity;

The current capacity of this document
- allow direct access to VValue[] length

property Count: integer read VCount;

Number of items stored in this document
- is 0 if Kind=dvUndefined
- is the number of name/value pairs for Kind=dvObject
- is the number of items for Kind=dvArray

property D[const aName: RawUTF8]: Double read GetDoubleByName write SetDoubleByName;

Direct access to a dvObject floating-point stored property value from its name
- slightly faster than the variant-based Value[] default property
- follows dvoNameCaseSensitive and dvoReturnNullForUnknownProperty options
- use GetAsDouble if you want to check the availability of the field
- D['prop'] := 1.23 would add a new property, or overwrite an existing

property I[const aName: RawUTF8]: Int64 read GetInt64ByName write SetInt64ByName;

Direct access to a dvObject Integer stored property value from its name
- slightly faster than the variant-based Value[] default property
- follows dvoNameCaseSensitive and dvoReturnNullForUnknownProperty options
- use GetAsInt/GetAsInt64 if you want to check the availability of the field
- I['prop'] := 123 would add a new property, or overwrite an existing

property Kind: TDocVariantKind read GetKind;

Returns the document internal layout
- just after initialization, it will return dvUndefined
- most of the time, you will add named values with AddValue() or by setting the variant properties: it will return dvObject
- but if you use AddItem(), values will have no associated names: the document will be a dvArray
- value computed from the dvoArray and dvoObject presence in Options

property Names: TRawUTF8DynArray read VName;

Direct access to the low-level internal array of names
- is void (nil) if Kind is not dvObject
- transtyping a variant and direct access to TDocVariantData is the fastest way of accessing all properties of a given dvObject:

```pascal
with TDocVariantData(aVariantObject) do
for i := 0 to Count-1 do
  writeln(Names[i], '=' , Values[i]);
```
property O[const aName: RawUTF8]: PDocVariantData read GetObjectExistingByName;  
  Direct access to a dvObject existing dvObject property from its name  
  - follows dvoNameCaseSensitive and dvoReturnNullForUnknownProperty options  
  - O['prop'] would return a fake void TDocVariant if the property is not existing or not a dvObject,  
    just like GetAsDocVariantSafe()  
  - use O_[‘prop’] to force adding any missing property

property Options: TDocVariantOptions read VOptions write SetOptions;  
  How this document will behave  
  - those options are set when creating the instance  
  - dvoArray and dvoObject are not options, but define the document Kind, so those items are  
    ignored when assigned to this property

property O_[const aName: RawUTF8]: PDocVariantData read GetObjectOrAddByName;  
  Direct access or add a dvObject's dvObject property from its name  
  - follows dvoNameCaseSensitive and dvoReturnNullForUnknownProperty options  
  - O_[‘prop’] would add a new property if there is none existing, or overwrite an existing property  
    which is not a dvObject

property S[const aName: RawUTF8]: string read GetStringByName write SetStringByName;  
  Direct string access to a dvObject UTF-8 stored property value from its name  
  - just a wrapper around U[] property, to avoid a compilation warning when using plain string  
    variables (internally, RawUTF8 will be used for storage)  
  - slightly faster than the variant-based Value[] default property  
  - follows dvoNameCaseSensitive and dvoReturnNullForUnknownProperty options  
  - use GetAsRawUTF8() if you want to check the availability of the field  
  - S['prop'] := 'value' would add a new property, or overwrite an existing

property U[const aName: RawUTF8]: RawUTF8 read GetRawUTF8ByName write SetRawUTF8ByName;  
  Direct access to a dvObject UTF-8 stored property value from its name  
  - slightly faster than the variant-based Value[] default property  
  - follows dvoNameCaseSensitive and dvoReturnNullForUnknownProperty options  
  - use GetAsRawUTF8() if you want to check the availability of the field  
  - U['prop'] := 'value' would add a new property, or overwrite an existing
property Value[const aNameOrIndex: Variant]: Variant read GetValueOrItem write SetValueOrItem;

Find an item in this document, and returns its value
- raise an EDocVariant if aNameOrIndex is neither an integer nor a string
- raise an EDocVariant if Kind is dvArray and aNameOrIndex is a string or if Kind is dvObject and aNameOrIndex is an integer
- raise an EDocVariant if Kind is dvObject and if aNameOrIndex is a string, which is not found within the object property names and dvoReturnNullForUnknownProperty is set in Options
- raise an EDocVariant if Kind is dvArray and if aNameOrIndex is an integer, which is not within 0..Count-1 and dvoReturnNullForUnknownProperty is set in Options
- so you can use directly:
  // for an array document:
aVariant := TDocVariant.NewArray(['one',2,3,0]);
  for i := 0 to TDocVariantData(aVariant).Count-1 do
    aValue := TDocVariantData(aVariant).Value[i];
  // for an object document:
aVariant := TDocVariant.NewObject(['name','John','year',1972]);
  assert(aVariant.Name=TDocVariantData(aVariant)['name']);
  assert(aVariant.year=TDocVariantData(aVariant)['year']);
- due to the internal implementation of variant execution (somewhat slow _DispInvoke() function), it is bit faster to execute:
  aValue := TDocVariantData(aVariant).Value['name'];

instead of
  aValue := aVariant.name;

but of course, if want to want to access the content by index (typically for a dvArray), using Values[] - and Names[] - properties is much faster than this variant-indexed pseudo-property:
  with TDocVariantData(aVariant) do
    for i := 0 to Count-1 do
      Writeln(Values[i]);

is faster than:
  with TDocVariantData(aVariant) do
    for i := 0 to Count-1 do
      Writeln(Value[i]);

which is faster than:
  for i := 0 to aVariant.Count-1 do
    Writeln(aVariant._(i));

- this property will return the value as varByRef (just like with variant late binding of any TDocVariant instance), so you can write:
  var Doc: TDocVariantData; // stack-allocated variable
  begin
    Doc.InitJSON('{arr:[1,2]}');
    assert(Doc.Count=2);
    Doc.Value['arr'].Add(3); // works since Doc.Value['arr'] is varByRef
    writeln(Doc.ToJSON); // will write '{"arr":[1,2,3]}'
  end;

- if you want to access a property as a copy, i.e. to assign it to a variant variable which will stay alive after this TDocVariant instance is release, you should not use Value[] but rather GetValueOrRaiseException or GetValueOrNull/GetValueOrEmpty
- see U[] I[] B[] D[] O[] O_[] A[] A_[] _[] properties for direct access of strong typed values
property Values: TVariantDynArray read VValue;

Direct access to the low-level internal array of values
- transpanting a variant and direct access to TDocVariantData is the fastest way of accessing all properties of a given dvObject:
  ```delphi
  with TDocVariantData(aVariantObject) do
  for i := 0 to Count - 1 do
    writeln(Names[i], ' = ', Values[i]);
  ```
- or to access a dvArray items (e.g. a MongoDB collection):
  ```delphi
  with TDocVariantData(aVariantArray) do
  for i := 0 to Count - 1 do
    writeln(Values[i]);
  ```

property VarType: word read VType;

Return the custom variant type identifier, i.e. DocVariantType.VarType

property _[aIndex: integer]: PDocVariantData read GetAsDocVariantByIndex;

Direct access to a dvArray's TDocVariant property from its index
- simple values may directly use Values[] dynamic array, but to access a TDocVariantData members, this property is safer
- follows dvoReturnNullForUnknownProperty option to raise an exception
- _[n dx] would return a fake void TDocVariant if aIndex is out of range, if the property is not existing or not a TDocVariantData (just like GetAsDocVariantSafe)

TPrecisionTimer = object(TObject)

High resolution timer (for accurate speed statistics)
- WARNING: under Windows, this record MUST be aligned to 32-bit, otherwise iFreq=0 - so you can use TLocalPrecisionTimer/ILocalPrecisionTimer if you want to allocate a local timer instance on the stack

function ByCount(Count: QWord): TShort16;

Compute the time elapsed by count, with appened time resolution (us,ms,s)

function FromExternalQueryPerformanceCounters(const CounterDiff: QWord): QWord;

Delphi 2007 is buggy as hell low-level method to force values settings to allow thread safe timing
- by default, this timer is not thread safe: you can use this method to set the timing values from manually computed performance counters
- the caller should also use a mutex to prevent from race conditions: see e.g. TSynMonitor.FromExternalQueryPerformanceCounters implementation
- returns the time elapsed, in micro seconds (i.e. LastTime value)
- warning: Start, Stop, Pause and Resume methods are then disallowed

function LastTime: TShort16;

Textual representation of last process timing after counter stopped
- Time returns a total elapsed time, whereas this method only returns the latest resumed time
- with appened time resolution (us,ms,s) - from MicroSecToString()
- not to be used in normal code, but e.g. for custom performance analysis

function PerSec(const Count: QWord): QWord;

Compute the per second count
function ProfileCurrentMethod: IUnknown;

*Resume a paused timer until the method ends*
- will internally create a TInterfaceObject class to let the compiler generate a try..finally block as expected to call Pause at method ending
- is therefore very convenient to have consistent Resume/Pause calls
- for proper use, expect TPrecisionTimer to be initialized to 0 before execution (e.g. define it as a protected member of a class)
- typical use is to declare a fTimeElapsed: TPrecisionTimer protected member, then call fTimeElapsed.ProfileCurrentMethod at the beginning of all process expecting some timing, then log/save fTimeElapsed.Stop content
- FPC TIP: result should be assigned to a local variable of IUnknown type

function SizePerSec(Size: QWord): shortstring;
*Returns e.g. '16.9 MB in 102.20ms i.e. 165.5 MB/s'*

function Started: boolean;
*Check if Start/Resume were called at least once*

function Stop: TShort16;
*Stop the timer, returning the total time elapsed as text*
- with appended time resolution (us,ms,s) - from MicroSecToString()
- is just a wrapper around Pause + Time
- you can call Resume to continue adding time to this timer

function StopInMicroSec: TSynMonitorTotalMicroSec;
*Stop the timer, returning the total time elapsed as microseconds*
- is just a wrapper around Pause + Time
- you can call Resume to continue adding time to this timer

function Time: TShort16;
*Textual representation of total time elapsed*
- with appended time resolution (us,ms,s) - from MicroSecToString()
- not to be used in normal code (which could rather call the Stop method), but e.g. for custom performance analysis

procedure FromExternalMicroSeconds(const MicroSeconds: QWord);
*Low-level method to force values settings to allow thread safe timing*
- by default, this timer is not thread safe: you can use this method to set the timing values from manually computed performance counters
- the caller should also use a mutex to prevent from race conditions: see e.g. TSynMonitor.FromExternalMicroSeconds implementation
- warning: Start, Stop, Pause and Resume methods are then disallowed

procedure Init;
*Initialize the timer*
- will fill all internal state with 0
- not necessary e.g. if TPrecisionTimer is defined as a TObject field

procedure Pause;
*Stop the timer, ready to continue its time measurement via Resume*
- will also compute the global Time value
- do nothing if no previous Start/Resume call is pending
procedure Resume;
   
   Resume a paused timer, or start an initialized timer
   - do nothing if no timer has been initialized or paused just before
   - if the previous method called was Init, will act like Start
   - if the previous method called was Pause, it will continue counting

procedure Start;
   
   Initialize and start the high resolution timer
   - similar to Init + Resume

property LastTimeInMicroSec: TSynMonitorOneMicroSec read fLastTime write fLastTime;
   Timing in micro seconds of the last process
   - not to be used in normal code, but e.g. for custom performance analysis

property PauseCount: TSynMonitorCount read fPauseCount;
   How many times the Pause method was called, i.e. the number of tasks processed

property TimeInMicroSec: TSynMonitorTotalMicroSec read fTime write fTime;
   Time elapsed in micro seconds after counter stopped
   - not to be used in normal code, but e.g. for custom performance analysis

ILocalPrecisionTimer = interface(IInterface)
   Interface to a reference counted high resolution timer instance
   - implemented by TLocalPrecisionTimer

function ByCount(Count: cardinal): RawUTF8;
   Compute the time elapsed by count, with appended time resolution (us,ms,s)

function PerSec(Count: cardinal): cardinal;
   Compute the per second count

function Stop: TShort16;
   Stop the timer, returning the time elapsed, with appended time resolution (us,ms,s)

procedure Pause;
   Stop the timer, ready to continue its time measure

procedure Resume;
   Resume a paused timer, or start it if it hasn’t been started

procedure Start;
   Start the high resolution timer
TLocalPrecisionTimer = class(TInterfacedObject)
  Reference counted high resolution timer (for accurate speed statistics)
  - since TPrecisionTimer shall be 32-bit aligned, you can use this class to initialize a local
    auto-freeing ILocalPrecisionTimer variable on stack
  - to be used as such:
    var
      Timer: ILocalPrecisionTimer;
    (...) 
      Timer := TLocalPrecisionTimer.Create;
      Timer.Start;
    (...) 

constructor CreateAndStart;
  Initialize the instance, and start the high resolution timer

function ByCount(Count: cardinal): RawUTF8;
  Compute the time elapsed by count, with appened time resolution (us,ms,s)

function PerSec(Count: cardinal): cardinal;
  Compute the per second count

function Stop: TShort16;
  Stop the timer, returning the time elapsed, with appened time resolution (us,ms,s)

procedure Pause;
  Stop the timer, ready to continue its time measure

procedure Resume;
  Resume a paused timer, or start the timer

procedure Start;
  Start the high resolution timer

TSynMonitorTime = class(TSynPersistent)
  Able to serialize any cumulative timing as raw micro-seconds number or text
  - "cumulative" time would add each process value, e.g. SOA methods execution

function PerSecond(const Count: QWord): QWord;
  Compute a number per second, of the current value

property MicroSec: TSynMonitorTotalMicroSec read fMicroSeconds write fMicroSeconds;
  Delphi 2007 is buggy as hell micro seconds time elapsed, as raw number

property Text: TShort16 read GetAsText;
  Micro seconds time elapsed, as '... us-ns-ms-s' text

TSynMonitorOneTime = class(TSynPersistent)
  Able to serialize any immediate timing as raw micro-seconds number or text
  - "immediate" size won't accumulate, i.e. may be e.g. last process time
function PerSecond(const Count: QWord): QWord;
    Compute a number per second, of the current value

property MicroSec: TSynMonitorOneMicroSec read fMicroSeconds write fMicroSeconds;
    Delphi 2007 is buggy as hell micro seconds time elapsed, as raw number

property Text: TShort16 read GetAsText;
    Micro seconds time elapsed, as '... us-ns-ms-s' text

TSynMonitorSize = class(TSynMonitorSizeParent)
    Able to serialize any cumulative size as bytes number
    - "cumulative" time would add each process value, e.g. global IO consumption

property Bytes: TSynMonitorTotalBytes read fBytes write fBytes;
    Number of bytes, as raw number

property Text: TShort16 read GetAsText;
    Number of bytes, as '... B-KB-MB-GB' text

TSynMonitorOneSize = class(TSynMonitorSizeParent)
    Able to serialize any immediate size as bytes number
    - "immediate" size won't accumulate, i.e. may be e.g. computer free memory at a given time

property Bytes: TSynMonitorOneBytes read fBytes write fBytes;
    Number of bytes, as raw number

property Text: TShort16 read GetAsText;
    Number of bytes, as '... B-KB-MB-GB' text

TSynMonitorThroughput = class(TSynMonitorSizeParent)
    Able to serialize any bandwidth as bytes count per second
    - is usually associated with TSynMonitorOneSize properties, e.g. to monitor IO activity

property BytesPerSec: QWord read fBytesPerSec write fBytesPerSec;
    Number of bytes per second, as raw number

property Text: TShort16 read GetAsText;
    Number of bytes per second, as '... B-KB-MB-GB/s' text

TSynMonitor = class(TSynPersistentLock)
    A generic value object able to handle any task / process statistic
    - base class shared e.g. for ORM, SOA or DDD, when a repeatable data process is to be monitored
    - this class is thread-safe for its methods, but you should call explicitly Lock/UnLock to access its individual properties

InternalTimer: TPrecisionTimer;
    Low-level high-precision timer instance
constructor Create; overload; override;
Initialize the instance nested class properties

constructor Create(const aName: RawUTF8); reintroduce; overload; virtual;
Initialize the instance nested class properties
- you can specify identifier associated to this monitored resource which would be used for TSynMonitorUsage persistence

destructor Destroy; override;
Finalize the instance

function ComputeDetails: variant;
Returns a TDocVariant with all published properties information
- thread-safe method

function ComputeDetailsJSON: RawUTF8;
Returns a JSON content with all published properties information
- thread-safe method

function FromExternalQueryPerformanceCounters(const CounterDiff: QWord): QWord;
Used to allow thread safe timing
- by default, the internal TPrecisionTimer is not thread safe: you can use this method to update the timing from many threads
- if you use this method, ProcessStart, ProcessDoTask and ProcessEnd methods are disallowed, and the global fTimer won't be used any more
- will return the processing time, converted into micro seconds, ready to be logged if needed
- thread-safe method

procedure ComputeDetailsTo(W: TTextWriter); virtual;
Appends a JSON content with all published properties information
- thread-safe method

procedure FromExternalMicroSeconds(const MicroSecondsElapsed: QWord);
Used to allow thread safe timing
- by default, the internal TPrecisionTimer is not thread safe: you can use this method to update the timing from many threads
- if you use this method, ProcessStart, ProcessDoTask and ProcessEnd methods are disallowed, and the global fTimer won't be used any more
- thread-safe method

class procedure InitializeObjArray(var ObjArr; Count: integer); virtual;
Create Count instances of this actual class in the supplied ObjArr[]

procedure Lock;
Lock the instance for exclusive access
- needed only if you access directly the instance properties

procedure ProcessDoTask; virtual;
Should be called each time a pending task is processed
- will increase the TaskCount property
- thread-safe method
procedure ProcessEnd; virtual;
   // Should be called when the process stops, to pause the internal timer
   // - thread-safe method

procedure ProcessError(const info: variant); virtual;
   // Should be called when an error occurred
   // - typical use is with ObjectToVariantDebug(E,...) kind of information
   // - thread-safe method

procedure ProcessErrorFmt(const Fmt: RawUTF8; const Args: array of const);
   // Should be called when an error occurred
   // - just a wrapper around overloaded ProcessError(), so a thread-safe method

procedure ProcessErrorNumber(info: integer);
   // Should be called when an error occurred
   // - typical use is with a HTTP status, e.g. as ProcessError(Call.OutStatus)
   // - just a wrapper around overloaded ProcessError(), so a thread-safe method

procedure ProcessErrorRaised(E: Exception);
   // Should be called when an Exception occurred
   // - just a wrapper around overloaded ProcessError(), so a thread-safe method

procedure ProcessStart; virtual;
   // Should be called when the process starts, to resume the internal timer
   // - thread-safe method

procedure ProcessStartTask; virtual;
   // Should be called when the process starts, and a task is processed
   // - similar to ProcessStart + ProcessDoTask
   // - thread-safe method

procedure Sum(another: TSynMonitor);
   // Could be used to manage information average or sums
   // - thread-safe method calling LockedSum protected virtual method

procedure UnLock;
   // Release the instance for exclusive access
   // - needed only if you access directly the instance properties

property AverageTime: TSynMonitorOneTime read fAverageTime;
   // The time spent in average during any working process

property Errors: TSynMonitorCount read fInternalErrors;
   // How many errors did occur during the processing

property LastError: variant read fLastInternalError;
   // Information about the last error which occurred during the processing

property LastTime: TSynMonitorOneTime read fLastTime;
   // The time spend during the last task processing

property MaximalTime: TSynMonitorOneTime read fMaximalTime;
   // The highest time spent during any working process
property MinimalTime: TSynMonitorOneTime read fMinimalTime;
The lowest time spent during any working process

property Name: RawUTF8 read fName write fName;
An identifier associated to this monitored resource
- is used e.g. for TSynMonitorUsage persistence/tracking

property PerSec: QWord read fPerSec;
Average of how many tasks did occur per second

property Processing: boolean read fProcessing write fProcessing;
Indicates if this thread is currently working on some process

property TaskCount: TSynMonitorCount64 read fTaskCount write fTaskCount;
How many times the task was performed

property TotalTime: TSynMonitorTime read fTotalTime;
The whole time spend during all working process

TSynMonitorWithSize = class(TSynMonitor)
Handle generic process statistic with a processing data size and bandwith

constructor Create; override;
Initialize the instance nested class properties

destructor Destroy; override;
Finalize the instance

procedure AddSize(const Bytes: QWord);
Increase the internal size counter
- thread-safe method

property Size: TSynMonitorSize read fSize;
How many total data has been handled during all working process

property Throughput: TSynMonitorThroughput read fThroughput;
Data processing bandwith, returned as B/KB/MB per second

TSynMonitorInputOutput = class(TSynMonitor)
Handle generic process statistic with a incoming and outgoing processing data size and bandwith

constructor Create; override;
Initialize the instance nested class properties

destructor Destroy; override;
Finalize the instance

procedure AddSize(const Incoming, Outgoing: QWord);
Increase the internal size counters
- thread-safe method
property Input: TSynMonitorSize read fInput;
   How many data has been received

property InputThroughput: TSynMonitorThroughput read fInputThroughput;
   Incoming data processing bandwith, returned as B/KB/MB per second

property Output: TSynMonitorSize read fOutput;
   How many data has been sent back

property OutputThroughput: TSynMonitorThroughput read fOutputThroughput;
   Outgoing data processing bandwith, returned as B/KB/MB per second

TSynMonitorServer = class(TSynMonitorInputOutput)
   Could monitor a standard Server
   - including Input/Output statistics and connected Clients count

function AddCurrentRequestCount(diff: integer): integer;
   How many concurrent requests are currently processed
   - returns the updated number of requests
   - thread-safe method

function GetClientsCurrent: TSynMonitorOneCount;
   Retrieve the number of connected clients
   - thread-safe method

procedure ClientConnect;
   Update ClientsCurrent and ClientsMax
   - thread-safe method

procedure ClientDisconnect;
   Update ClientsCurrent and ClientsMax
   - thread-safe method

procedure ClientDisconnectAll;
   Update ClientsCurrent to 0
   - thread-safe method

property ClientsCurrent: TSynMonitorOneCount read fClientsCurrent;
   Current count of connected clients

property ClientsMax: TSynMonitorOneCount read fClientsMax;
   Max count of connected clients

property CurrentRequestCount: integer read fCurrentRequestCount;
   How many concurrent requests are currently processed
   - modified via AddCurrentRequestCount() in TSQLRestServer.URI()
TSynInterfacedObject = class(TObject)

An abstract ancestor, for implementing a custom TInterfacedObject like class
- by default, will do nothing: no instance would be retrieved by QueryInterface unless the
  VirtualQueryInterface protected method is overriden, and _AddRef/_Release methods would call
  VirtualAddRef and VirtualRelease pure abstract methods
- using this class will leverage the signature difference between Delphi and FPC, among all
  supported platforms
- the class includes a RefCount integer field

property RefCount: integer read fRefCount write fRefCount;
The associated reference count

TSynFPUException = class(TSynInterfacedObject)

A simple class which will set FPU exception flags for a code block
- using an IUnknown interface to let the compiler auto-generate a try..finally block statement to
  reset the FPU exception register
- to be used e.g. as such:
  begin
   TSynFPUException.ForLibraryCode;
   ... now FPU exceptions will be ignored
   ... so here it is safe to call external library code
  end; // now FPU exception will be reset as with standard Delphi
- it will avoid any unexpected invalid floating point operation in Delphi code, whereas it was in fact
  triggered in some external library code

constructor Create(Expected8087Flag: word); reintroduce;
  Internal constructor
  - do not call this constructor directly, but rather use ForLibraryCode/ForDelphiCode class
    methods
  - for cpu32 flags are $1372 for Delphi, or $137F for library (mask all exceptions)
  - for cpu64 flags are $1920 for Delphi, or $1FA0 for library (mask all exceptions)

class function ForDelphiCode: IUnknown;
  After this method call, all FPU exceptions will be enabled
  - this is the Delphi normal behavior
  - until the method finishes (a try..finally block is generated by the compiler), then FPU
    exceptions will be disabled again
  - you have to put this e.g. before running an Delphi code from a callback executed in an external
    libray
  - this method is thread-safe and re-entrant (by reference-counting)

class function ForLibraryCode: IUnknown;
  After this method call, all FPU exceptions will be ignored
  - until the method finishes (a try..finally block is generated by the compiler), then FPU
    exceptions will be reset into "Delphi" mode
  - you have to put this e.g. before calling an external library
  - this method is thread-safe and re-entrant (by reference-counting)

IAutoFree = interface(IInterface)
Interface for TAutoFree to register another TObject instance to an existing IAutoFree local variable
TAutoFree = class(TInterfacedObject)

Simple reference-counted storage for local objects
- be aware that it won't implement a full ARC memory model, but may be just used to avoid
  writing some try ... finally blocks on local variables
- use with caution, only on well defined local scope

constructor Create(const varObjPairs: array of pointer); reintroduce; overload;
  Initialize the TAutoFree class for several local variables
  - do not call this constructor, but class function Several() instead

constructor Create(var localVariable; obj: TObject); reintroduce; overload;
  Initialize the TAutoFree class for one local variable
  - do not call this constructor, but class function One() instead

destructor Destroy; override;
  Will finalize the associated TObject instances
- note that releasing the TObject instances won't be protected, so any exception here may
  induce a memory leak: use only with "safe" simple objects, e.g. mORMot's TSQLRecord

class function One(var localVariable; obj: TObject): IAutoFree;
  Protect one local TObject variable instance life time
  - for instance, instead of writing:
    var myVar: TMyClass;
    begin
      myVar := TMyClass.Create;
      try
        ... use myVar
        finally
          myVar.Free;
        end;
      end;
    - you may write:
      var myVar: TMyClass;
      begin
        TAutoFree.One(myVar,TMyClass.Create);
        ... use myVar
      end; // here myVar will be released
  - warning: under FPC, you should assign the result of this method to a local IAutoFree variable -
    see bug http://bugs.freepascal.org/view.php?id=26602

class function Several(const varObjPairs: array of pointer): IAutoFree;
  Protect several local TObject variable instances life time
  - specified as localVariable/objectInstance pairs
  - you may write:
    var var1,var2: TMyClass;
    begin
      TAutoFree.Several([@var1,TMyClass.Create,
                       @var2,TMyClass.Create]);
      ... use var1 and var2
    end; // here var1 and var2 will be released
  - warning: under FPC, you should assign the result of this method to a local IAutoFree variable -
    see bug http://bugs.freepascal.org/view.php?id=26602
procedure Another(var localVariable; obj: TObject);

Protect another TObject variable to an existing IAutoFree instance life time
- you may write:
  var var1,var2: TMyClass;
  auto: IAutoFree;
  begin
    auto := TAutoFree.One(var1,TMyClass.Create),
    .... do something
    auto.Another(var2,TMyClass.Create);
    .... use var1 and var2
  end; // here var1 and var2 will be released

IAutoLocker = interface(IInterface)
An interface used by TAutoLocker to protect multi-thread execution

function ProtectMethod: IUnknown;
Will enter the mutex until the IUnknown reference is released
- using an IUnknown interface to let the compiler auto-generate a try..finally block statement to release the lock for the code block
- could be used as such under Delphi:
  begin
    ...
    // unsafe code
    fSharedAutoLocker.ProtectMethod;
    ...
    // thread-safe code
  end; // Local hidden IUnknown will release the lock for the method

- warning: under FPC, you should assign its result to a local variable - see bug
  http://bugs.freepascal.org/view.php?id=26602
  var LockFPC: IUnknown;
  begin
    ...
    // unsafe code
    LockFPC := fSharedAutoLocker.ProtectMethod;
    ...
    // thread-safe code
  end; // LockFPC will release the lock for the method

Or
  begin
    ...
    // unsafe code
    with fSharedAutoLocker.ProtectMethod do begin
      ...
      // thread-safe code
    end; // local hidden IUnknown will release the lock for the method
  end;

function Safe: PSynLocker;
Gives an access to the internal low-level TSynLocker instance used
procedure Enter;

Enter the mutex
- any call to Enter should be ended with a call to Leave, and protected by a try..finally block, as such:
begin
  ... // unsafe code
  fSharedAutoLocker.Enter;
  try
    ... // thread-safe code
  finally
    fSharedAutoLocker.Leave;
  end;
end;

procedure Leave;

Leave the mutex
- any call to Leave should be preceded with a call to Enter

TAutoLocker = class(TInterfacedObjectWithCustomCreate)

Reference-counted block code critical section
- you can use one instance of this to protect multi-threaded execution
- the main class may initialize a IAutoLocker property in Create, then call IAutoLocker.ProtectMethod in any method to make its execution thread safe
- this class inherits from TInterfacedObjectWithCustomCreate so you could define one published property of a mORMot.pas' TInjectableObject as IAutoLocker so that this class may be automatically injected
- you may use the inherited TAutoLockerDebug class, as defined in SynLog.pas, to debug unexpected race conditions due to such critical sections
- consider inherit from high-level TSynPersistentLock or call low-level fSafe := NewSynLocker / fSafe^.DoneAndFreemem instead

constructor Create; override;
  Initialize the mutex

destructor Destroy; override;
  Finalize the mutex
function ProtectMethod: IUnknown;

Will enter the mutex until the IUnknown reference is released
- as expected by IAutoLocker interface
- could be used as such under Delphi:

begin
    ... // unsafe code
    fSharedAutoLocker.ProtectMethod;
    ... // thread-safe code
end; // Local hidden IUnknown will release the lock for the method

- warning: under FPC, you should assign its result to a local variable - see bug
http://bugs.freepascal.org/view.php?id=26602

var LockFPC: IUnknown;
begin
    ... // unsafe code
    LockFPC := fSharedAutoLocker.ProtectMethod;
    ... // thread-safe code
end; // LockFPC will release the lock for the method

Or
begin
    ... // unsafe code
    with fSharedAutoLocker.ProtectMethod do begin
        ... // thread-safe code
    end; // Local hidden IUnknown will release the lock for the method
end;

function Safe: PSynLocker;

Access to the locking methods of this instance
- as expected by IAutoLocker interface

procedure Enter; virtual;

Enter the mutex
- as expected by IAutoLocker interface
- any call to Enter should be ended with a call to Leave, and protected by a try..finally block, as such:

begin
    ... // unsafe code
    fSharedAutoLocker.Enter;
    try
        ... // thread-safe code
    finally
        fSharedAutoLocker.Leave;
    end;
end;

procedure Leave; virtual;

Leave the mutex
- as expected by IAutoLocker interface

property Locker: TSynLocker read fSafe;

Direct access to the locking methods of this instance
- faster than IAutoLocker.Safe function
ILockedDocVariant = interface(IInterface)

  Internal error C3517 under Delphi 5 : ref-counted interface for thread-safe access to a TDocVariant document
  - is implemented e.g. by TLockedDocVariant, for IoC/DI resolution
  - fast and safe storage of any JSON-like object, as property/value pairs, or a JSON-like array, as values

function AddExistingProp(const Name: RawUTF8; var Obj: variant): boolean;
  Add an existing property value to the given TDocVariant document object
  - returns TRUE and add the Name/Value pair to Obj if Name is existing
  - returns FALSE if Name is not existing in the stored document
  - this method would use a lock during the Name lookup, but would always release the lock, even if returning FALSE (see AddExistingPropOrLock)

function AddExistingPropOrLock(const Name: RawUTF8; var Obj: variant): boolean;
  Add an existing property value to the given TDocVariant document object
  - returns TRUE and add the Name/Value pair to Obj if Name is existing, using an internal lock for thread-safety
  - returns FALSE if Name is not existing in the stored document, and lock the internal storage: caller should eventually release the lock via AddNewPropAndUnlock()
  - could be used as such, for implementing a thread-safe cache:

  ```delphi
  if not cache.AddExistingPropOrLock('Articles',Scope) then
    cache.AddNewPropAndUnlock('Articles',GetArticlesFromDB,Scope);
  ```

  here GetArticlesFromDB would occur inside the main lock

function Copy: variant;
  Makes a thread-safe copy of the internal TDocVariant document object or array

function Exists(const Name: RawUTF8; out Value: Variant): boolean;
  Check and return a given property by name
  - returns TRUE and fill Value with the value associated with the supplied Name, using an internal lock for thread-safety
  - returns FALSE if the Name was not found, releasing the internal lock: use ExistsOrLock() if you want to add the missing value

function ExistsOrLock(const Name: RawUTF8; out Value: Variant): boolean;
  Check and return a given property by name
  - returns TRUE and fill Value with the value associated with the supplied Name, using an internal lock for thread-safety
  - returns FALSE and set the internal lock if Name does not exist: caller should then release the lock via ReplaceAndUnlock()

function ToJSON(HumanReadable: boolean=false): RawUTF8;
  Save the stored values as UTF-8 encoded JSON Object

procedure AddItem(const Value: variant);
  Append a value to the internal TDocVariant document array
  - you should not use this method in conjunction with other document-based alternatives, like Exists/AddExistingPropOrLock or AddExistingProp
procedure AddNewProp(const Name: RawUTF8; const Value: variant; var Obj: variant);
  Add a property value to the given TDocVariant document object
  - this method would not expect the resource to be locked when called, as with
  AddNewPropAndUnlock
  - will use the internal lock for thread-safety
  - if the Name is already existing, would update/change the existing value
  - could be used as such, for implementing a thread-safe cache:
    if not cache.AddExistingProp('Articles',Scope) then
      cache.AddNewProp('Articles',GetArticlesFromDB,Scope);
  here GetArticlesFromDB would occur outside the main lock

procedure AddNewPropAndUnlock(const Name: RawUTF8; const Value: variant; var Obj: variant);
  Add a property value to the given TDocVariant document object and to the internal stored document, then release a previous lock
  - call of this method should have been precedeed by AddExistingPropOrLock() returning false, i.e. be executed on a locked instance

procedure Clear;
  Delete all stored properties

procedure ReplaceAndUnlock(const Name: RawUTF8; const Value: Variant; out LocalValue: Variant);
  Set a value by property name, and set a local copy
  - could be used as such, for implementing a thread-safe cache:
    if not cache.ExistsOrLock('prop',local) then
      cache.ReplaceAndUnlock('prop',newValue,local);
  - call of this method should have been precedeed by ExistsOrLock() returning false, i.e. be executed on a locked instance

property Value[const Name: RawUTF8]: Variant read GetValue write SetValue;
  The document fields would be safely accessed via this property
  - this is the main entry point of this storage
  - will raise an EDocVariant exception if Name does not exist at reading
  - implementation class would make a thread-safe copy of the variant value

TLockedDocVariant = class(TInterfacedObjectWithCustomCreate)
  Allows thread-safe access to a TDocVariant document
  - this class inherits from TInterfacedObjectWithCustomCreate so you could define one published property of a mORMot.pas' TInjectableObject as ILockedDocVariant so that this class may be automatically injected

  constructor Create(options: TDocVariantOptions); reintroduce; overload;
    Initialize the thread-safe document storage with the corresponding options

  constructor Create(FastStorage: boolean); reintroduce; overload;
    Initialize the thread-safe document storage
constructor Create; overload; override;

*Initialize the thread-safe document with a fast TDocVariant*
- i.e. call Create(true) aka Create(JSON_OPTIONS[true])
- will be the TInterfacedObjectWithCustomCreate default constructor, called e.g. during IoC/DI resolution

**destructor** Destroy; override;

*Finalize the storage*

**function** AddExistingProp(const Name: RawUTF8; var Obj: variant): boolean;

*Add an existing property value to the given TDocVariant document object*
- returns TRUE and add the Name/Value pair to Obj if Name is existing
- returns FALSE if Name is not existing in the stored document
- this method would use a lock during the Name lookup, but would always release the lock, even if returning FALSE (see AddExistingPropOrLock)

**function** AddExistingPropOrLock(const Name: RawUTF8; var Obj: variant): boolean;

*Add an existing property value to the given TDocVariant document object*
- returns TRUE and add the Name/Value pair to Obj if Name is existing
- returns FALSE if Name is not existing in the stored document

**function** Copy: variant;

*Makes a thread-safe copy of the internal TDocVariant document object or array*

**function** Exists(const Name: RawUTF8; out Value: Variant): boolean;

*Check and return a given property by name*

**function** ExistsOrLock(const Name: RawUTF8; out Value: Variant): boolean;

*Check and return a given property by name*
- this version

**function** ToJSON(HumanReadable: boolean=false): RawUTF8;

*Save the stored value as UTF-8 encoded JSON Object*
- implemented as just a wrapper around VariantSaveJSON()

**procedure** AddItem(const Value: variant);

*Append a value to the internal TDocVariant document array*

**procedure** AddNewProp(const Name: RawUTF8; const Value: variant; var Obj: variant);

*Add a property value to the given TDocVariant document object*
- this method would not expect the resource to be locked when called, as with AddNewPropAndUnlock
- will use the internal lock for thread-safety
- if the Name is already existing, would update/change the existing value

**procedure** AddNewPropAndUnlock(const Name: RawUTF8; const Value: variant; var Obj: variant);

*Add a property value to the given TDocVariant document object and to the internal stored document*

**procedure** Clear;

*Delete all stored properties*
procedure ReplaceAndUnlock(const Name: RawUTF8; const Value: Variant; out LocalValue: Variant);
    Set a value by property name, and set a local copy

property Value[const Name: RawUTF8]: Variant read GetValue write SetValue;
    The document fields would be safely accessed via this property
    - will raise an EDocVariant exception if Name does not exist
    - result variant is returned as a copy, not as varByRef, since a copy will definitively be more
      thread safe

Types implemented in the SynCommons unit

PBlock128 = ^TBlock128;
    Pointer to a 128-bit buffer

PBooleanArray = ^TBooleanArray;
    Redefine here with {$R-}

PDALen = PPtrInt;
    Internal pointer integer type used for dynamic array header length field

PDocVariantData = ^TDocVariantData;
    Pointer to a TDocVariant storage
    - since variants may be stored by reference (i.e. as varByRef), it may be a good idea to use such a
      pointer via DocVariantData(aVariant)^ or _Safe(aVariant)^ instead of TDocVariantData(aVariant), if
      you are not sure how aVariant was allocated (may be not _Obj/_Json)

PDocVariantOptions = ^TDocVariantOptions;
    Pointer to a set of options for a TDocVariant storage
    - you may use e.g. @JSON_OPTIONS[true], @JSON_OPTIONS[false],
      @JSON_OPTIONS_FAST_STRICTJSON or @JSON_OPTIONS_FAST_EXTENDED

PDWordRec = ^TDWordRec;
    Points to the binary of an unsigned 32-bit value

PDynArray = ^TDynArray;
    A pointer to a TDynArray wrapper instance

PDynArrayHasher = ^TDynArrayHasher;
    Pointer to a TDynArrayHasher instance

PFileName = ^TFileName;
    A pointer to a TFileName variable

PHash128 = ^THash128;
    Pointer to a 128-bit hash value

PHash128Array = ^THash128Array;
    Pointer to an infinite array of 128-bit hash values

PHash128Rec = ^THash128Rec;
    Pointer to 128-bit hash map variable record

PHash160 = ^THash160;
    Pointer to a 160-bit hash value
PHash192 = ^THash192;
   Pointer to a 192-bit hash value

PHash256 = ^THash256;
   Pointer to a 256-bit hash value

PHash256Array = ^THash256Array;
   Pointer to an infinite array of 256-bit hash values

PHash256Rec = ^THash256Rec;
   Pointer to 256-bit hash map variable record

PHash384 = ^THash384;
   Pointer to a 384-bit hash value

PHash512 = ^THash512;
   Pointer to a 512-bit hash value

PHash512Array = ^THash512Array;
   Pointer to an infinite array of 512-bit hash values

PHash512Rec = ^THash512Rec;
   Pointer to 512-bit hash map variable record

PPatchCode = ^TPatchCode;
   Pointer to a small memory buffer used to backup a RedirectCode() hook

PPointerClassHashed = ^TPPointerClassHashed;
   A reference to a TPointerClassHashed instance

PPPrecisionTimer = ^PPPrecisionTimer;
   Indirect reference to a pointer to a high resolution timer object/rec

PPPPrecisionTimer = ^TPPrecisionTimer;
   Pointer to a high resolution timer object/rec

PPtrInt = ^PtrInt;
   A CPU-dependent signed integer type cast of a pointer of pointer
   - used for 64-bit compatibility, native under Free Pascal Compiler

PPtrUInt = ^PtrUInt;
   A CPU-dependent unsigned integer type cast of a pointer of pointer
   - used for 64-bit compatibility, native under Free Pascal Compiler

PQWord = ^QWord;
   Points to an unsigned Int64

PQWordRec = ^TQWordRec;
   Points to the binary of an unsigned 64-bit value

PRawByteString = ^RawByteString;
   Pointer to a RawByteString

PRefCnt = ^TRefCnt;
   Internal pointer integer type used for string/dynarray header reference counters

PStrLen = ^TStrLen;
Internal pointer integer type used for string header length field

PSynDate = ^TSynDate;
   A pointer to a TSynDate instance

PSynMonitor = ^TSynMonitor;
   References a TSynMonitor instance

PSynNameValue = ^TSynNameValue;
   A reference pointer to a Name/Value RawUTF8 pairs storage

PTimeLogBits = ^TTimeLogBits;
   Pointer to a memory structure for direct access to a TTimeLog type value

PtrInt = integer;
   A CPU-dependent signed integer type cast of a pointer / register
   - used for 64-bit compatibility, native under Free Pascal Compiler

PtrUInt = cardinal;
   A CPU-dependent unsigned integer type cast of a pointer / register
   - used for 64-bit compatibility, native under Free Pascal Compiler

PUnixMSTime = ^TUnixMSTime;
   Pointer to a timestamp stored as millisecond-based Unix Time

PUnixTime = ^TUnixTime;
   Pointer to a timestamp stored as second-based Unix Time

PUTF8Char = type PAnsiChar;
   A simple wrapper to UTF-8 encoded zero-terminated PAnsiChar
   - PAnsiChar is used only for Win-Ansi encoded text
   - the Synopse mORMot framework uses mostly this PUTF8Char type, because all data is internally stored and expected to be UTF-8 encoded

PWord = System.PWord;
   Redefined here to not use the wrong definitions from Windows.pas

QWord = type Int64;
   Unsigned Int64 doesn’t exist under older Delphi, but is defined in FPC
   - and UInt64 is buggy as hell under Delphi 2007 when inlining functions: older compilers will fallback to signed Int64 values
   - anyway, consider using SortDynArrayQWord() to compare QWord values in a safe and efficient way, under a CPUX86
   - you may use UInt64 explicitly in your computation (like in SynEcc.pas), if you are sure that Delphi 6-2007 compiler handles your code as expected, but mORMot code will expect to use QWord for its internal process (e.g. ORM/SOA serialization)

RawByteString = type AnsiString;
   Define RawByteString, as it does exist in Delphi 2009+
   - to be used for byte storage into an AnsiString
   - use this type if you don't want the Delphi compiler not to do any code page conversions when you assign a typed AnsiString to a RawByteString, i.e. a RawUTF8 or a WinAnsiString

RawJSON = type RawUTF8;
   RawJSON will indicate that this variable content would stay in raw JSON
- i.e. won't be serialized into values
- could be any JSON content: number, string, object or array
- e.g. interface-based service will use it for efficient and AJAX-ready transmission of TSQLTableJSON result

<table>
<thead>
<tr>
<th>RawUnicode</th>
<th>type</th>
<th>AnsiString</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>RawUnicode</em> is an <em>Unicode String stored in an AnsiString</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- faster than WideString, which are allocated in Global heap (for COM)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- an AnsiChar(#0) is added at the end, for having a true WideChar(#0) at ending</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- length(RawUnicode) returns memory bytes count: use (length(RawUnicode) shr 1) for WideChar count (that's why the definition of this type since Delphi 2009 is AnsiString(1200) and not UnicodeString)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- pointer(RawUnicode) is compatible with Win32 'Wide' API call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- mimic Delphi 2009 UnicodeString, without the WideString or Ansi conversion overhead</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- all conversion to/from AnsiString or RawUTF8 must be explicit: the compiler is not able to make valid implicit conversion on CP_UTF16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RawUTF8</th>
<th>type</th>
<th>AnsiString</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>RawUTF8</em> is an <em>UTF-8 String stored in an AnsiString</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- use this type instead of System.UTF8String, which behavior changed between Delphi 2009 compiler and previous versions: our implementation is consistent and compatible with all versions of Delphi compiler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- mimic Delphi 2009 UTF8String, without the charset conversion overhead</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- all conversion to/from AnsiString or RawUnicode must be explicit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SynUnicode</th>
<th>WideString</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>SynUnicode</em> is the fastest available Unicode native string type, depending on the compiler used*</td>
<td></td>
</tr>
<tr>
<td>- this type is native to the compiler, so you can use Length() Copy() and such functions with it (this is not possible with RawUnicodeString type)</td>
<td></td>
</tr>
<tr>
<td>- before Delphi 2009+, it uses slow OLE compatible WideString (with our Enhanced RTL, WideString allocation can be made faster by using an internal caching mechanism of allocation buffers) - WideString allocation has been made much faster since Windows Vista/Seven)</td>
<td></td>
</tr>
<tr>
<td>- starting with Delphi 2009, it uses fastest UnicodeString type, which allow Copy On Write, Reference Counting and fast heap memory allocation</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TAlgoCompressLoad</th>
<th>( aclNormal, aclSafeSlow, aclNoCrcFast )</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Deprecated TRawUTF8MethodList should be replaced by a TSynDictionary define the implemetation used by TAlgoCompress.Decompress()</em></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TBits32</th>
<th>set of 0..31;</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Fast access to 32-bit integer bits</em></td>
<td></td>
</tr>
<tr>
<td>- the compiler will generate bt/btr/bts opcodes</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TBits64</th>
<th>set of 0..63;</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Fast access to 64-bit integer bits</em></td>
<td></td>
</tr>
<tr>
<td>- the compiler will generate bt/btr/bts opcodes</td>
<td></td>
</tr>
<tr>
<td>- as used by GetBit64/SetBit64/UnSetBit64</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TBits8</th>
<th>set of 0..7;</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Fast access to 8-bit integer bits</em></td>
<td></td>
</tr>
<tr>
<td>- the compiler will generate bt/btr/bts opcodes</td>
<td></td>
</tr>
</tbody>
</table>

| TBlock128       | array[0..3] of cardinal; |
**Store a 128-bit buffer**
- e.g. an AES block
- consumes 16 bytes of memory

```
TChar64 = array[0..63] of AnsiChar;
```

**Some stack-allocated zero-terminated character buffer**
- as used by GetNextTChar64

```
TCharConversionFlags = set of (ccfNoTrailingZero, ccfReplacementCharacterForUnmatchedSurrogate);
```

Option set for RawUnicodeToUtf8() conversion

```
TDatetimeMS = type TDateTime;
```

A type alias, which will be serialized as ISO-8601 with milliseconds
- i.e. 'YYYY-MM-DD hh:mm:ss.sss' or 'YYYYMMDD hhmmss.sss' format

```
TDatetimeMSDynArray = array of TDateTimeMS;
```

A dynamic array of TDateTimeMS values

```
TDocVariantKind = (dvUndefined, dvObject, dvArray);
```

Define the TDocVariant storage layout
- if it has one or more named properties, it is a dvObject
- if it has no name property, it is a dvArray

```
TDocVariantOption = (dvoIsArray, dvoIsObject, dvoNameCaseSensitive, dvoCheckForDuplicatedNames, dvoReturnNullForUnknownProperty, dvoValueCopiedByReference, dvoJSONParseDoNotTryCustomVariants, dvoJSONObjectParseWithinString, dvoSerializeAsExtendedJson, dvoAllowDoubleValue, dvoInternNames, dvoInternValues);
```

Possible options for a TDocVariant JSON/BSON document storage
- dvIsArray and dvIsObject will store the "Kind: TDocVariantKind" state - you should never have to define these two options directly
- dvNameCaseSensitive will be used for every name lookup - here case-insensitivity is restricted to a-z A-Z 0-9 and _ characters
- dvCheckForDuplicatedNames will be used for method TDocVariantData.AddValue(), but not when setting properties at variant level: for consistency, "aVariant.AB := aValue" will replace any previous value for the name "AB"
- dvReturnNullForUnknownProperty will be used when retrieving any value from its name (for dvObject kind of instance), or index (for dvArray or dvObject kind of instance)
- by default, internal values will be copied by-value from one variant instance to another, to ensure proper safety - but it may be too slow: if you set dvoValueCopiedByReference, the internal TDocVariantData.VValue/VName instances will be copied by-reference, to avoid memory allocations, but it may break internal process if you change some values in place (since VValue/VName and VCount won't match) - as such, if you set this option, ensure that you use the content as read-only
- any registered custom types may have an extended JSON syntax (e.g. TBSONVariant does for MongoDB types), and will be searched during JSON parsing, unless dvoJSONParseDoNotTryCustomVariants is set (slightly faster)
- by default, it will only handle direct JSON [array] of {object}: but if you define dvoJSONObjectParseWithinString, it will also try to un-escape a JSON string first, i.e. handle "[array]" or "{object}" content (may be used e.g. when JSON has been retrieved from a database TEXT column) - is used for instance by VariantLoadJSON()
- JSON serialization will follow the standard layout, unless dvoSerializeAsExtendedJson is set so that the property names would not be escaped with double quotes, writing '{name:"John",age:123}'
instead of '{"name":"John","age":123}': this extended json layout is compatible with
http://docs.mongodb.org/manual/reference/mongodb-extended-json and with TDocVariant JSON
unserialization, also our SynCrossPlatformJSON unit, but NOT recognized by most JSON clients, like
AJAX/JavaScript or C#/Java
- by default, only integer/Int64/currency number values are allowed, unless dvoAllowDoubleValue is
set and 32-bit floating-point conversion is tried, with potential loss of precision during the
conversion
- dvoInternNames and dvoInternValues will use shared TRawUTF8Interning instances to maintain a
list of RawUTF8 names/values for all TDocVariant, so that redundant text content will be allocated
only once on heap

```
TDocVariantOptions = set of TDocVariantOption;
Set of options for a TDocVariant storage
- you can use JSON_OPTIONS[true] if you want to create a fast by-reference local document as with
  _ObjFast/_ArrFast/_JsonFast
  - i.e. [dvoReturnNullForUnknownProperty,dvoValueCopiedByReference]
- when specifying the options, you should not include dvoArray nor dvoObject directly in the set,
  but explicitly define TDocVariantDataKind

TDynArrayAfterLoadFrom = procedure(var A) of object;
Optional event called by TDynArray.LoadFrom method after each item load
- could be used e.g. for string interning or some custom initialization process
- won't be called if the dynamic array has ElemType=nil

TDynArrayHashOne = function(const Elem; Hasher: THasher): cardinal;
Function prototype to be used for hashing of a dynamic array element
- this function must use the supplied hasher on the Elem data

TDynArrayJSONCustomReader = function(P: PUTF8Char; var aValue; out aValid: Boolean ;
CustomVariantOptions: PDocVariantOptions): PUTF8Char of object;
Method prototype for custom unserialization of a dynamic array item
- each element of the dynamic array will be called as aValue parameter of this callback
- can be used also at record level, if the record has a type information (i.e. shall contain a managed
type within its fields)
- to be used with TTextWriter.RegisterCustomJSONSerializer() method
- implementation code could call e.g. GetJSONField() low-level function, and returns a pointer to the
last handled element of the JSON input buffer, as such (aka EndOfBuffer variable as expected by:GetJSONField):
  var V: TFV absolute aValue;
  begin
    (...)
    V.Detailed := UTF8ToString(GetJSONField(P,P));
    if P=nil then
      exit;
    aValid := true;
    result := P; // ',' or ']' for last item of array
  end;
- implementation code shall follow the same exact format for the associated
TDynArrayJSONCustomWriter callback

TDynArrayJSONCustomWriter = procedure(const aWriter: TTextWriter; const aValue) of object;
Method prototype for custom serialization of a dynamic array item
- each element of the dynamic array will be called as aValue parameter of this callback

- can be used also at record level, if the record has a type information (i.e. shall contain a managed type within its fields)
- to be used with TTextWriter.RegisterCustomJSONSerializer() method
- note that the generated JSON content will be appended after a ']' and before a '[' as a normal JSON array, but each item can be any JSON structure (i.e. a number, a string, but also an object or an array)
- implementation code could call aWriter.Add/AddJSONEscapeString...
- implementation code shall follow the same exact format for the associated TDynArrayJSONCustomReader callback

TDynArrayKind = ( djNone, djBoolean, djByte, djWord, djInteger, djCardinal, djSingle, djInt64, djQWord, djDouble, djCurrency, djTimeLog, djDateTime, djDateTimeMS, djRawUTF8, djWinAnsi, djString, djRawByteString, djWideString, djSynUnicode, djHash128, djHash512, djInterface, djVariant, djCustom );

Internal enumeration used to specify some standard Delphi arrays
- will be used e.g. to match JSON serialization or TDynArray search (see TDynArray and TDynArrayHash InitSpecific method)
- djBoolean would generate an array of JSON boolean values
- djByte .. djTimeLog match numerical JSON values
- djDateTime .. djHash512 match textual JSON values
- djVariant will match standard JSON serialization (including TDocVariant or other custom types, if any)
- djCustom will be used for registered JSON serializer (invalid for InitSpecific methods call)
- see also djPointer and djObject constant aliases for a pointer or TObject field hashing / comparison
- is used also by TDynArray.InitSpecific() to define the main field type

TDynArrayKinds = set of TDynArrayKind;
Internal set to specify some standard Delphi arrays

TDynArraySortCompare = function(const A,B): integer;
Function prototype to be used for TDynArray Sort and Find method
- common functions exist for base types: see e.g. SortDynArrayBoolean, SortDynArrayByte, SortDynArrayWord, SortDynArrayInteger, SortDynArrayCardinal, SortDynArrayInt64, SortDynArrayQWord, SortDynArraySingle, SortDynArrayDouble, SortDynArrayAnsiString, SortDynArrayAnsiStringl, SortDynArrayUnicodeString, SortDynArrayUnicodeStringl, SortDynArrayString, SortDynArrayStringl
- any custom type (even records) can be compared then sort by defining such a custom function
- must return 0 if A=B, -1 if A<B, 1 if A>B

TEventDynArrayHashOne = function(const Elem): cardinal of object;
Event handler to be used for hashing of a dynamic array element
- can be set as an alternative to TDynArrayHashOne

TEventDynArraySortCompare = function(const A,B): integer of object;
Event oriented version of TDynArraySortCompare

TFindFilesDynArray = array of TFindFiles;
Result list, as returned by FindFiles()

TFloatNan = ( fnNumber, fnNan, fnInf, fnNegInf );
The non-number values potentially stored in an IEEE floating point

TFloatType = ( ftSingle, ftDoub, ftExtended, ftComp, ftCurr );
Specify floating point (ftFloat) storage size and precision
- here ftDouble is renamed ftDoub to avoid confusion with TSQLDBFieldType

TGUIDShortString = string[38];

Stack-allocated ASCII string, used by GUIDToShort() function

THash128 = array[0..15] of byte;

Store a 128-bit hash value
- e.g. a MD5 digest, or array[0..3] of cardinal (TBlock128)
- consumes 16 bytes of memory

THash128Array = array[0..(maxInt div SizeOf(THash128))-1] of THash128;

Map an infinite array of 128-bit hash values
- each item consumes 16 bytes of memory

THash128DynArray = array of THash128;

Store several 128-bit hash values
- e.g. MD5 digests
- consumes 16 bytes of memory per item

THash160 = array[0..19] of byte;

Store a 160-bit hash value
- e.g. a SHA-1 digest
- consumes 20 bytes of memory

THash192 = array[0..23] of byte;

Store a 192-bit hash value
- consumes 24 bytes of memory

THash256 = array[0..31] of byte;

Store a 256-bit hash value
- e.g. a SHA-256 digest, a TECCSignature result, or array[0..7] of cardinal
- consumes 32 bytes of memory

THash256Array = array[0..(maxInt div SizeOf(THash256))-1] of THash256;

Map an infinite array of 256-bit hash values
- each item consumes 32 bytes of memory

THash256DynArray = array of THash256;

Store several 256-bit hash values
- e.g. SHA-256 digests, TECCSignature results, or array[0..7] of cardinal
- consumes 32 bytes of memory per item

THash384 = array[0..47] of byte;

Store a 384-bit hash value
- e.g. a SHA-384 digest
- consumes 48 bytes of memory

THash512 = array[0..63] of byte;

Store a 512-bit hash value
- e.g. a SHA-512 digest, a TECCSignature result, or array[0..15] of cardinal
- consumes 64 bytes of memory

THash512Array = array[0..(maxInt div SizeOf(THash512))-1] of THash512;

Map an infinite array of 512-bit hash values
- each item consumes 64 bytes of memory

```pascal
THash512DynArray = array of THash512;
```

*Store several 512-bit hash values*
- e.g. SHA-512 digests, or array[0..15] of cardinal
- consumes 64 bytes of memory per item

```pascal
THasher = function (crc: cardinal; buf: PAnsiChar; len: cardinal): cardinal;
```

*Function prototype to be used for hashing of an element*
- it must return a cardinal hash, with as less collision as possible
- TDynArrayHashed.Init will use crc32c() if no custom function is supplied, which will run either as software or SSE4.2 hardware, with good collisions for most used kind of data

```pascal
TIntelCpuFeature = ( cfFPU, cfVME, cfDE, cfPSE, cfTSC, cfMSR, cfPAE, cfMCE, cfCX8,
 cfAPIC, cf_d10, cfSEP, cfMTRR, cfPGE, cfMCA, cfCMOV, cfPAT, cfPSE36, cfPSN, cfCLFSH,
 cf_d20, cfDS, cfACPI, cfMMX, cfFXSR, cfSSE, cfSSE2, cfSS, cfHTT, cfTM, cfIA64, cfPBE,
 cfSSE3, cfCLMUL, cfDS64, cfMON, cfSDSCPL, cfVMPX, cfSMX, cfEST, cfTME, cfSSE3, cfC1D,
 cfSDTBG, cfFMA, cfCX16, cfXTRR, cfPDECM, cf_c16, cfPCID, cfDCAC, cfSSE41, cfSSE42, cfX2A,
 cfVMXE, cfXPOPCNT, cfTSC2, cfAESN, cfXSA, cfOSXS, cfAVX, cfF16C, cfCRAND, cfHYP, cfCSSG,
 cf_b01, cfCGX, cfBMM1, cfHLE, cfAVX2, cf_b06, cfSMEP, cfBMI2, cfERMS, cfINVPID, cfRTM,
 cfPQM, cf_b13, cfFMX, cfFPE, cfAVX512F, cfAVX512QD, cfRDSEED, cfADAX, cfFSMAP,
 cfAVX512FMA, cfFPCM, cfCLFLUSH, cfCMM, cfFPT, cfAVX512PF, cfAVX512ER, cfAVX512CD,
 cfSHA, cfAVX512BW, cfAVX512VL, cfPREFW1, cfAVX512VBM, cfUMIP, cfPKU, cfOSPAE,
 cf_c05, cfAVX512VMI2, cf_c07, cfSNI, cfVSAES, cfVCLMUL, cfAVX512DNI, cfAVX52BITA, cf_c13,
 cfAVX512VI, cf_c15, cf_c16, cf_c17, cf_c18, cf_c19, cf_c20, cf_c21, cfROPID, cf_c23,
 cf_c24, cf_c25, cf_c26, cf_c27, cf_c28, cf_c29, cfSGX, cf_c31, cf_d0, cf_d1,
 cfAVX512NMA, cfAVX512MAS, cf_d4, cf_d5, cf_d6, cf_d7 );
```

*The potential features, retrieved from an Intel CPU*
- see https://en.wikipedia.org/wiki/CPUID#EAX.3D1:_Processor_Info_and_Feature_Bits
- is defined on all platforms, since an ARM desktop could browse Intel logs

```pascal
TIntelCpuFeatures = set of TIntelCpuFeature;
All features, as retrieved from an Intel CPU
```

```pascal
TInterfacedObjectClass = class of TInterfacedObject;
Class-reference type (metaclass) of a TInterfacedObject
```

```pascal
TInterfacedObjectWithCustomCreateClass = class of TInterfacedObjectWithCustomCreate;
Used to determine the exact class type of a TInterfacedObjectWithCustomCreate
- could be used to create instances using its virtual constructor
```

```pascal
TJsonChar = set of (jcJsonIdentifierFirstChar, jcJsonIdentifier, jcEndOfJSONField,
 jcEndOfJSONFieldOr0, jcEndOfJSONValueField, jcDigitChar, jcDigitFirstChar,
 jcDigitFloatChar);
```

*Kind of character used from JSON_CHARS[] for efficient JSON parsing*

```pascal
TJsonCharSet = array[AnsiChar] of TJsonChar;
Defines a branch-less table used for JSON parsing
```

```pascal
TJSONCustomParserCustomSimpleKnownType = ( ktNone, ktEnumeration, ktSet, ktGUID,
 ktFixedArray, ktStaticArray, ktDynamicArray, ktBinary );
```

*Which kind of property does TJSONCustomParserCustomSimple refer to*

```pascal
TJSONCustomParserRTTIExpectedEnd = ( eeNothing, eeSquare, eeCurly, eeEndKeyWord );
```

*How an RTTI expression is expected to finish*
TJSONCustomParserRTTI = array of TJSONCustomParserRTTI;

An array of RTTI properties information
- we use dynamic arrays, since all the information is static and we do not need to remove any RTTI information

TJSONCustomParserRTTIType = ( ptArray, ptBoolean, ptByte, ptCardinal, ptCurrency, ptDouble, ptExtended, ptInt64, ptInteger, ptQWord, ptRawByteString, ptRawJSON, ptRawUTF8, ptRecord, ptSingle, ptString, ptSynUnicode, ptDateTime, ptDateTimeMS, ptGUID, ptID, ptTimeLog, ptVariant, ptWideString, ptWord, ptCustom );
The kind of variables handled by TJSONCustomParser
- the last item should be ptCustom, for non simple types

TJSONCustomParserSerializationOption = ( soReadIgnoreUnknownFields, soWriteHumanReadable, soCustomVariantCopiedByReference, soWriteIgnoreDefault );
How TJSONCustomParser would serialize/unserialize JSON content

TJSONCustomParserSerializationOptions = set of TJSONCustomParserSerializationOption;
How TJSONCustomParser would serialize/unserialize JSON content
- by default, during reading any unexpected field will stop and fail the process - if soReadIgnoreUnknownFields is defined, such properties will be ignored (can be very handy when parsing JSON from a remote service)
- by default, JSON content will be written in its compact standard form, ready to be parsed by any client - you can specify soWriteHumanReadable so that some line feeds and indentation will make the content more readable
- by default, internal TDocVariant variants will be copied by-value from one instance to another, to ensure proper safety - but it may be too slow: if you set soCustomVariantCopiedByReference, any internal TDocVariantData.VValue/VName instances will be copied by-reference, to avoid memory allocations, BUT it may break internal process if you change some values in place (since VValue/VName and VCount won't match) - as such, if you set this option, ensure that you use the content as read-only
- by default, all fields are persistented, unless soWriteIgnoreDefault is defined and void values (e.g. "" or 0) won't be written
- you may use TTextWriter.RegisterCustomJSONSerializerSetOptions() class method to customize the serialization for a given type

TLogEscape = array[0..LOGESCAPELEN*3+5] of AnsiChar;
Buffer to be allocated on stack when using LogEscape()

TMultiPartDynArray = array of TMultiPart;
Used by MultiPartFormDataDecode() to return all its data items

TNameValuePUTF8CharDynArray = array of TNameValuePUTF8Char;
Used e.g. by JSONDecode() overloaded function to returns name/value pairs

TObjectListLocked = TSynObjectListLocked;
Deprecated class name, for backward compatibility only

TObjectListPropertyHashedAccessProp = function(aObject: TObject): pointer;
Function prototype used to retrieve a pointer to the hashed property value of a TObjectListPropertyHashed list

TOnKeyNotify = procedure(Sender: TObject; const Key: RawUTF8) of object;
Event signature to notify a given string key
TOnKeyResolve = function(const aInterface: TGUID; const Key: RawUTF8; out Obj): boolean of object;

- Event signature to locate a service for a given string key
- used e.g. by TRawUTF8ObjectCacheList.OnKeyResolve property

TOnNotifySortedIntegerChange = procedure(const Sender; Value: integer) of object;

- Event handler called by NotifySortedIntegerChanges()
- Sender is an opaque const value, maybe a TObject or any pointer

TOnReducePerItem = function(Item: PDocVariantData): boolean of object;

- Method used by TDocVariantData.ReduceAsArray to filter each object
- should return TRUE if the item match the expectations

TOnReducePerValue = function(const Value: variant): boolean of object;

- Method used by TDocVariantData.ReduceAsArray to filter each object
- should return TRUE if the item match the expectations

TOnStringTranslate = procedure (var English: string) of object;

- A generic callback, which can be used to translate some text on the fly
- maps procedure TLanguageFile.Translate(var English: string) signature as defined in mORMoti18n.pas
- can be used e.g. for TSynMustache's {{"English text"}} callback

TOnSynNameValueConvertRawUTF8 = function(const text: RawUTF8): RawUTF8 of object;

- Event handler used to convert on the fly some UTF-8 text content

TOnSynNameValueNotify = procedure(const Item: TSynNameValueItem; Index: PtrInt) of object;

- Callback event used by TSynNameValue

TOnTextWriterEcho = function(Sender: TTextWriter; Level: TSynLogInfo; const Text: RawUTF8): boolean of object;

- Callback used to echo each line of TTextWriter class
- should return TRUE on success, FALSE if the log was not echoed: but TSynLog will continue logging, even if this event returned FALSE

TOnTextWriterFlush = procedure(Text: PUTF8Char; Len: PtrInt) of object;

- Event signature for TTextWriter.OnFlushToStream callback

TOnTextWriterObjectProp = function(Sender: TTextWriter; Value: TObject; PropInfo: pointer; Options: TTextWriterWriteObjectOptions): boolean of object;

- Callback used by TTextWriter.WriteObject to customize class instance serialization
- should return TRUE if the supplied property has been written (including the property name and the ending ',' character), and doesn't need to be processed with the default RTTI-based serializer

TOnValueGreater = function(IndexA, IndexB: PtrInt): boolean of object;

- Comparison function as expected by MedianQuickSelect()
- should return TRUE if Values[IndexA]>Values[IndexB]

TOperatingSystem = ( osUnknown, osWindows, osLinux, osOSX, osBSD, osPOSIX, osArch, osAurox, osDebian, osFedora, osGentoo, osKnoppix, osMint, osMandrake, osMandriva, osNovell, osUbuntu, osSlackware, osSolaris, osSuse, osSynology, osTrustix, osClear, osUnite, osRedHat, osLFS, osOracle, osMageia, osCentOS, osCloud, osXen, osAmazon, osCoreOS, osAlpine, osAndroid );

- The recognized operating systems
- it will also recognize some Linux distributions

TOrdType = ( otSByte, otUByte, otSWord, otUWord, otSLong, otULong );
  Specify ordinal (tkInteger and tkEnumeration) storage size and sign
  - note: Int64 is stored as its own TTypeKind, not as tkInteger

TPatchCode = array[0..4] of byte;
  Small memory buffer used to backup a RedirectCode() redirection hook

TPersistentWithCustomCreateClass = class of TPersistentWithCustomCreate;
  Used to determine the exact class type of a TPersistentWithCustomCreateClass
  - could be used to create instances using its virtual constructor

TPublishedMethodInfoDynArray = array of TPublishedMethodInfo;
  Information about all methods, as returned by GetPublishedMethods

TPUTF8CharArray = array[0..MaxInt div SizeOf(PUTF8Char)-1] of PUTF8Char;
  A Row/Col array of PUTF8Char, for containing sqlite3_get_table() result

TPUTF8CharDynArray = array of PUTF8Char;
  A dynamic array of PUTF8Char pointers

TRawUTF8DynArray = array of RawUTF8;
  A dynamic array of UTF-8 encoded strings

TRawUTF8ListFlags = set of ( fObjectsOwned, fCaseSensitive, fNoDuplicate, fOnChangeTriggered);
  Possible values used by TRawUTF8List.Flags

TRawUTF8ListLocked = type TRawUTF8List;
  Some declarations used for backward compatibility only

TRefCnt = longint;
  Internal integer type used for string/dynarray header reference counters

TShort16 = string[16];
  Used e.g. by PointerToHexShort/CardinalToHexShort/Int64ToHexShort/FormatShort16
  - such result type would avoid a string allocation on heap, so are highly recommended e.g. when logging small pieces of information

TShort4 = string[4];
  Used e.g. by UInt4DigitsToShort/UInt3DigitsToShort/UInt2DigitsToShort
  - such result type would avoid a string allocation on heap

TStreamClass = class of Tstream;
  Class-reference type (metaclass) of a TStream

TStrLen = longint;
  Internal integer type used for string header length field

TSynAnsicharSet = set of AnsiChar;
  Used to store a set of 8-bit encoded characters

TSynByteSet = set of Byte;
  Used to store a set of 8-bit unsigned integers

TSynDateDynArray = array of TSynDate;
Store several dates as Year/Month/Day

```pascal
TSynDictionaryCanDeleteEvent = function(const aKey, aValue; aIndex: integer): boolean of object;
    Event called by TSynDictionary.DeleteDeprecate
    - called just before deletion: return false to by-pass this item

TSynDictionaryEvent = function(const aKey; var aValue; aIndex, aCount: integer; aOpaque: pointer): boolean of object;
    Event called by TSynDictionary.ForEach methods to iterate over stored items
    - if the implementation method returns TRUE, will continue the loop
    - if the implementation method returns FALSE, will stop values browsing
    - aOpaque is a custom value specified at ForEach() method call

TSynDictionaryInArray = ( iaFind, iaFindAndDelete, iaFindAndUpdate, iaFindAndAddIfNotExisting, iaAdd );
    Internal flag, used only by TSynDictionary.InArray protected method

TSynExtended = extended;
    The floating-point type to be used for best precision and speed
    - will allow to fallback to double e.g. on x64 and ARM CPUs

TSynInvokeableVariantTypeClass = class of TSynInvokeableVariantType;
    Class-reference type (metaclass) of custom variant type definition
    - used by SynRegisterCustomVariantType() function

TSynLogExceptionToStr = function(WR: TTextWriter; const Context: TSynLogExceptionContext): boolean;
    Global hook callback to customize exceptions logged by TSynLog
    - should return TRUE if all needed information has been logged by the event handler
    - should return FALSE if Context.EAddr and Stack trace is to be appended

TSynLogInfo = ( sllNone, sllInfo, sllDebug, sllTrace, sllWarning, sllError, sllEnter, sllLeave, sllLastError, sllException, sllExceptionOS, sllMemory, sllStackTrace, sllFail, sllSQL, sllCache, sllResult, sllDB, sllHTTP, sllClient, sllServer, sllServiceCall, sllServiceReturn, sllUserAuth, sllCustom1, sllCustom2, sllCustom3, sllCustom4, sllNewRun, sllDDDError, sllDDDInfo, sllMonitoring );
    The available logging events, as handled by TSynLog
    - defined in SynCommons so that it may be used with TTextWriter.AddEndOfLine
    - sllInfo will log general information events
    - sllDebug will log detailed debugging information
    - sllTrace will log low-level step by step debugging information
    - sllWarning will log unexpected values (not an error)
    - sllError will log errors
    - sllEnter will log every method start
    - sllLeave will log every method exit
    - sllLastError will log the GetLastError OS message
    - sllException will log all exception raised - available since Windows XP
    - sllExceptionOS will log all OS low-level exceptions (EDivByZero, ERangeError, EAccessViolation...)
    - sllMemory will log memory statistics
    - sllStackTrace will log caller's stack trace (it's by default part of TSynLogFamily.LevelStackTrace like sllError, sllException, sllExceptionOS, sllLastError and sllFail)
    - sllFail was defined for TSynTestsLogged.Failed method, and can be used to log some customer-side assertions (may be notifications, not errors)
```
- sllSQL is dedicated to trace the SQL statements
- sllCache should be used to trace the internal caching mechanism
- sllResult could trace the SQL results, JSON encoded
- sllDB is dedicated to trace low-level database engine features
- sllHTTP could be used to trace HTTP process
- sllClient/sllServer could be used to trace some Client or Server process
- sllServiceCall/sllServiceReturn to trace some remote service or library
- sllUserAuth to trace user authentication (e.g. for individual requests)
- sllCustom* items can be used for any purpose
- sllNewRun will be written when a process opens a rotated log
- sllDDDError will log any DDD-related low-level error information
- sllDDDInfo will log any DDD-related low-level debugging information
- sllMonitoring will log the statistics information (if available), or may be used for real-time chat among connected people to ToolsAdmin

```pascal
TSynLogInfoDynArray = array of TSynLogInfo;

A dynamic array of logging event levels

TSynLogInfos = set of TSynLogInfo;

Used to define a set of logging level abilities
- i.e. a combination of none or several logging event
- e.g. use LOG_VERBOSE constant to log all events, or LOG_STACKTRACE to log all errors and exceptions

TSynMonitorBytesPerSec = type QWord;

Would identify the process throughput, during monitoring
- it indicates e.g. "immediate" bandwidth usage
- any property defined with this type would be identified by TSynMonitorUsage

TSynMonitorClass = class of TSynMonitor;

Class-reference type (metaclass) of a process statistic information

TSynMonitorCount = type cardinal;

Would identify a cumulative number of processes, during monitoring
- any property defined with this type would be identified by TSynMonitorUsage

TSynMonitorCount64 = type QWord;

Would identify a cumulative number of processes, during monitoring
- any property defined with this type would be identified by TSynMonitorUsage

TSynMonitorInputOutputObjArray = array of TSynMonitorInputOutput;

A list of incoming/outgoing data process statistics

TSynMonitorObjArray = array of TSynMonitor;

A list of simple process statistics

TSynMonitorOneBytes = type QWord;

Would identify an immediate process information as bytes count, during monitoring
- "immediate" size won't accumulate, i.e. may be e.g. computer free memory at a given time
- any property defined with this type would be identified by TSynMonitorUsage

TSynMonitorOneCount = type cardinal;

Would identify an immediate time count information, during monitoring
- "immediate" counts won't accumulate, e.g. may store the current number of thread used by a
process
- any property defined with this type would be identified by TSynMonitorUsage

TSynMonitorOneMicroSec = type QWord;

Would identify an immediate time process information in micro seconds, during monitoring
- "immediate" time won't accumulate, i.e. may store the duration of the latest execution of a SOA computation
- any property defined with this type would be identified by TSynMonitorUsage

TSynMonitorTotalBytes = type QWord;

Would identify a process information as cumulative bytes count, during monitoring
- "cumulative" size would add some byte for each process, e.g. input/output
- any property defined with this type would be identified by TSynMonitorUsage

TSynMonitorTotalMicroSec = type QWord;

Would identify a cumulative time process information in micro seconds, during monitoring
- "cumulative" time would add each process timing, e.g. for statistics about SOA computation of a given service
- any property defined with this type would be identified by TSynMonitorUsage

TSynMonitorType = ( smvUndefined, smvOneMicroSec, smvOneBytes, smvOneCount, smvBytesPerSec, smvMicroSec, smvBytes, smvCount, smvCount64 );

The kind of value stored in a TSynMonitor / TSynMonitorUsage property
- i.e. match TSynMonitorTotalMicroSec, TSynMonitorOneMicroSec, TSynMonitorOneCount, TSynMonitorOneBytes, TSynMonitorBytesPerSec, TSynMonitorTotalBytes, TSynMonitorCount and TSynMonitorCount64 types as used to store statistic information
- "cumulative" values would sum each process values, e.g. total elapsed time for SOA execution, task count or total I/O bytes
- "immediate" (e.g. svOneBytes or smvBytesPerSec) values would be an evolving single value, e.g. an average value or current disk free size
- use SYNMONITORVALUE_CUMULATIVE = [smvMicroSec, smvBytes, smvCount, smvCount64] constant to identify the kind of value
- TSynMonitorUsage.Track() would use MonitorPropUsageValue() to guess the tracked properties type from class RTTI

TSynMonitorTypes = set of TSynMonitorType;

Value types as stored in TSynMonitor / TSynMonitorUsage

TSynMonitorWithSizeObjArray = array of TSynMonitorWithSize;

A list of data process statistics

TSynNameValueItemDynArray = array of TSynNameValueItem;

Name/Value pairs storage, as used by TSynNameValue class

TSynPersistentClass = class of TSynPersistent;

Used to determine the exact class type of a TSynPersistent
- could be used to create instances using its virtual constructor

TSynPersistentLockClass = class of TSynPersistentLock;

Class-reference type (metaclass) of an TSynPersistentLock class

TSynPersistentLockDynArray = array of TSynPersistentLock;

Abstract dynamic array of TSynPersistentLock instance
- note defined as T*ObjArray, since it won't
TSystemPath = ( spCommonData, spUserData, spCommonDocuments, spUserDocuments, 
spTempFolder, spLog );

Identify an operating system folder

TTextChar = set of (tcNot01013, tc1013, tcCtrlNotLF, tcCtrlNot0Comma, tcWord, 
tcIdentifierFirstChar, tcIdentifier, tcURIUnreserved);

Char categories for text line/word/identifiers/uri parsing

TTextWriterClass = class of TTextWriterWithEcho;

Class of our simple TEXT format writer to a Stream, with echoing
- as used by TSynLog for writing its content
- see TTextWriterWithEcho.SetAsDefaultJSONClass

TTextWriterHTMLFormat = ( hfNone, hfAnyWhere, hfOutsideAttributes, hfWithinAttributes );

The potential places were TTextWriter.AddHtmlEscape should process proper HTML string escaping, 
unless hfNone is used
  &lt; &gt; " -> &lt; &gt; &amp; &quot;
by default (hfAnyWhere)
  &lt; &gt; " -> &lt; &gt; &amp; &quot;
outside HTML attributes (hfOutsideAttributes)
  &lt; &gt; " -> &lt; &gt; &amp; &quot;
within HTML attributes (hfWithinAttributes)

TTextWriterJSONFormat = ( jsonCompact, jsonHumanReadable, jsonUnquotedPropName, 
jsonUnquotedPropNameCompact );

The available JSON format, for TTextWriter.AddJSONReformat() and its JSONBufferReformat() and 
JSONReformat() wrappers
- jsonCompact is the default machine-friendly single-line layout
- jsonHumanReadable will add line feeds and indentation, for a more human-friendly result
- jsonUnquotedPropName will emit the jsonHumanReadable layout, but with all property names 
  being quoted only if necessary: this format could be used e.g. for configuration files - this format, 
similar to the one used in the MongoDB extended syntax, is not JSON compatible: do not use it e.g. 
with AJAX clients, but is would be handled as expected by all our units as valid JSON input, without 
previous correction
- jsonUnquotedPropNameCompact will emit single-line layout with unquoted property names

TTextWriterKind = ( twNone, twJSONEscape, twOnSameLine );

Kind of adding in a TTextWriter

TTextWriterOption = ( twoStreamIsOwned, twoFlushToStreamNoAutoResize, 
twoEnumSetsAsTextInRecord, twoEnumSetsAsBooleanInRecord, twoFullSetsAsStar, 
twoTrimLeftEnumSets, twoForceJSONExtended, twoForceJSONStandard, twoEndOfLineCRLF, 
twoBufferIsExternal, twoIgnoreDefaultInRecord );

Available global options for a TTextWriter instance
- TTextWriter.WriteObject() method behavior would be set via their own 
  TTextWriterWriteObjectOptions, and work in conjunction with those settings
- twoStreamIsOwned would be set if the associated TStream is owned by the TTextWriter instance
- twoFlushToStreamNoAutoResize would forbid FlushToStream to resize the internal memory buffer 
  when it appears undersized - FlushFinal will set it before calling a last FlushToStream
- by default, custom serializers defined via RegisterCustomJSONSerializer() would let 
  AddRecordJSON() and AddDynArrayJSON() write enumerates and sets as integer numbers, unless 
  twoEnumSetsAsTextInRecord or twoEnumSetsAsBooleanInRecord (exclusively) are set - for
Mustache data context, twoEnumSetsAsBooleanInRecord will return a JSON object with "setname":true/false fields
- variants and nested objects would be serialized with their default JSON serialization options, unless twoForceJSONExtended or twoForceJSONStandard is defined
- when enumerates and sets are serialized as text into JSON, you may force the identifiers to be left-trimmed for all their lowercase characters (e.g. sIError -> 'Error') by setting twoTrimLeftEnumSets
- this option would default to the global TTextWriter.SetDefaultEnumTrim setting
- twoEndOfLineCRLF would reflect the TTextWriter.EndOfLineCRLF property
- twoBuffersIsExternal would be set if the temporary buffer is not handled by the instance, but specified at constructor, maybe from the stack
- twoIgnoreDefaultInRecord will force custom record serialization to avoid writing the fields with default values, i.e. enable soWriteIgnoreDefault when TJSONCustomParserRTTI.WriteOneLevel is called

TTextWriterOptions = set of TTextWriterOption;

Options set for a TTextWriter instance
- allows to override e.g. AddRecordJSON() and AddDynArrayJSON() behavior; or set global process customization for a TTextWriter

TTextWriterStackBuffer = array[0..8191] of AnsiChar;
May be used to allocate on stack a 8KB work buffer for a TTextWriter
- via the TTextWriter.CreateOwnedStream overloaded constructor

TTextWriterWriteObjectOption = ( woHumanReadable, woDontStoreDefault, woFullExpand, woStoreClass, woStorePointer, woStoreStoredFalse, woHumanReadableFullSetsAsStar, woHumanReadableEnumSetsAsComment, woEnumSetsAsText, woDateTimeWithMagic, woDateTimeWithZSuffix, woTimeLogAsText, woIDAsIDstr, woSQLRawBlobAsBase64, woHideSynPersistentPassword, woObjectListWontStoreClassName, woDontStoreEmptyString, woDontStoreInherited, woInt64AsHex, woDontStore0 );

Available options for TTextWriter.WriteObject() method
- woHumanReadable will add some line feeds and indentation to the content, to make it more friendly to the human eye
- woDontStoreDefault (which is set by default for WriteObject method) will avoid serializing properties including a default value (JSONToObject function will set the default values, so it may help saving some bandwidth or storage)
- woFullExpand will generate a debugger-friendly layout, including instance class name, sets/enumerates as text, and reference pointer - as used by TSynLog and ObjectToJSONFull()
- woStoreClassName will add a "ClassName":"TMyClass" field
- woStorePointer will add a "Address":"0431298A" field, and .map/.mab source code line number corresponding to ESynException.RaisedAt
- woStoreStoredFalse will write the 'stored false' properties, even if they are marked as such (used e.g. to persist all settings on file, but disallow the sensitive - password - fields be logged)
- woHumanReadableFullSetsAsStar will store an human-readable set with all its enumerates items set to be stored as ['*']
- woHumanReadableEnumSetAsComment will add a comment at the end of the line, containing all available values of the enumeration or set, e.g:
"Enum": "Destroying", // Idle,Started,Finished,Destroying
- woEnumSetsAsText will store sets and enumerables as text (is also included in woFullExpand or woHumanReadable)
- woDateTimeWithMagic will append the JSON_SQLDATE_MAGIC (i.e. U+FFF1) before the ISO-8601 encoded TDateTime value
- woDateTimeWithZSuffix will append the Z suffix to the ISO-8601 encoded TDateTime value, to
identify the content as strict UTC value
- TTTimeLog would be serialized as Int64, unless woTimeLogAsText is defined
- since TSQLRecord.ID could be huge Int64 numbers, they may be truncated on client side, e.g. to
53-bit range in JavaScript: you could define woIDAsIDstr to append an additional
"ID_str": "###########" field
- by default, TSQLRawBlob properties are serialized as null, unless woSQLRawBlobAsBase64 is defined
- if woHideSynPersistentPassword is set, TSynPersistentWithPassword.Password field will be
serialized as "***" to prevent security issues (e.g. in log)
- by default, TObjectList will set the woStoreClassName for its nested objects, unless
woObjectListWontStoreClassNa name is defined
- void strings would be serialized as "", unless woDontStoreEmptyString is defined so that such
properties would not be written
- all inherited properties would be serialized, unless woDontStoreInherited is defined, and only the
topmost class level properties would be serialized
- woInt64AsHex will force Int64/QWord to be written as hexadecimal string - see j2oAllowInt64Hex
reverse option fot Json2Object
- woDontStore0 will avoid serializing number properties equal to 0

TTextWriterWriteObjectOptions = set of TTextWriterWriteObjectOption;

Options set for TTextWriter.WriteObject() method

TThreadID = cardinal;

Used to store the handle of a system Thread

TTTimeLog = type Int64;

Fast bit-encoded date and time value
- faster than ISO-8601 text and TDateTime, e.g. can be used as published property field in mORMot's
TSQLRecord (see also TModTime and TCreateTime)
- use internally for computation an abstract "year" of 16 months of 32 days of 32 hours of 64
minutes of 64 seconds - same as Iso8601ToTimeLog()
- use TimeLogFromDateTime/TimeLogToDateTIme/TimeLogNow functions, or type-cast any
TTTimeLog value with the TTTimeLogBits memory structure for direct access to its bit-oriented content
(or via PTimeLogBits pointer)
- since TTTimeLog type is bit-oriented, you can't just add or subtract two TTTimeLog values when
doing date/time computation: use a TDateTime temporary conversion in such case:

aTimestamp := TimeLogFromDateTime(IncDay(TimeLogToDateTime(aTimestamp)));
**Synopse mORMot Framework**

Software Architecture Design 1.18

Date: September 16, 2020

---

e.g. its tkLString)

```pascal
TUnixMSTime = type Int64;

  *Timestamp stored as millisecond-based Unix Time*
  - i.e. the number of milliseconds since 1970-01-01 00:00:00 UTC
  - see TUnixTime for a second resolution Unix Timestamp
  - use UnixMSTimeToDate/Time/TimeToUnixMSTime functions to convert it to/from a regular
    DateTime
  - also one of the JavaScript date encodings

TUnixMSTimeDynArray = array of TUnixMSTime;

  *Dynamic array of timestamps stored as millisecond-based Unix Time*

TUnixTime = type Int64;

  *Timestamp stored as second-based Unix Time*
  - i.e. the number of seconds since 1970-01-01 00:00:00 UTC
  - is stored as 64-bit value, so that it won't be affected by the "Year 2038" overflow issue
  - see TUnixMSTime for a millisecond resolution Unix Timestamp
  - use UnixTimeToDate/TimeToUnixTime functions to convert it to/from a regular
    DateTime
  - use UnixTimeUTC to return the current timestamp, using fast OS API call
  - also one of the encodings supported by SQLite3 date/time functions

TUnixTimeDynArray = array of TUnixTime;

  *Dynamic array of timestamps stored as second-based Unix Time*

TUTF8Compare = function(P1,P2: PUTF8Char): PtrInt;

  *Function prototype used internally for UTF-8 buffer comparison*
  - used in mORMot.pas unit during TSQLTable rows sort and by TSQLQuery

TValuePUTF8CharArray = array[0..maxInt div SizeOf(TValuePUTF8Char)-1] of
  TValuePUTF8Char;

  *Used e.g. by JSONDecode/overloaded function to returns values*

TVarDataStaticArray = array[0..MaxInt div SizeOf(TVarData)-1] of TVarData;

  *A TVarData values array*
  - is not called TVarDataArray to avoid confusion with the corresponding type already defined in
    Variants.pas, and used for custom late-binding

TVariantCompare = function(const V1,V2: variant): PtrInt;

  *Function prototype used internally for variant comparison*
  - used in mORMot.pas unit e.g. by TDocVariantData.SortByValue

TWindowsVersion = ( wUnknown, w2000, wXP, wXP_64, wServer2003, wServer2003_R2, wVista,
  wVista_64, wServer2008, wServer2008_64, wSeven, wSeven_64, wServer2008_R2,
  wServer2008_R2_64, wEight, wEight_64, wServer2012, wServer2012_64, wEightOne,
  wEightOne_64, wServer2012R2, wServer2012R2_64, wTen, wTen_64, wServer2016,
  wServer2016_64, wServer2019_64);

  *The recognized Windows versions*
  - defined even outside MSWINDOWS to allow process e.g. from monitoring tools

unaligned = Double;

  *Will actually change anything only on FPC ARM/Aarch64 platforms*

WinAnsiString = type AnsiString;
```
WinAnsiString is a WinAnsi-encoded AnsiString (code page 1252)
- use this type instead of System.String, which behavior changed between Delphi 2009 compiler and previous versions: our implementation is consistent and compatible with all versions of Delphi compiler
- all conversion to/from RawUTF8 or RawUnicode must be explicit

Constants implemented in the SynCommons unit

ALGO_SAFE: array[boolean] of TAlgoCompressLoad = (aclNormal, aclSafeSlow);
Used e.g. as when ALGO_SAFE[SafeDecompression] for TAlgoCompress.Decompress

ALLBITS_CARDINAL: array[1..32] of Cardinal = (1 shl 1-1, 1 shl 2-1, 1 shl 3-1, 1 shl 4-1, 1 shl 5-1, 1 shl 6-1, 1 shl 7-1, 1 shl 8-1, 1 shl 9-1, 1 shl 10-1, 1 shl 11-1, 1 shl 12-1, 1 shl 13-1, 1 shl 14-1, 1 shl 15-1, 1 shl 16-1, 1 shl 17-1, 1 shl 18-1, 1 shl 19-1, 1 shl 20-1, 1 shl 21-1, 1 shl 22-1, 1 shl 23-1, 1 shl 24-1, 1 shl 25-1, 1 shl 26-1, 1 shl 27-1, 1 shl 28-1, 1 shl 29-1, 1 shl 30-1, $7fffffff, $ffffffff);
Constant array used by GetAllBits() function (when inlined)

BINARYCONTENT_TYPE = 'application/octet-stream';
MIME content type used for raw binary data

BINARYCONTENT_TYPE_HEADER = HEADERCONTENT_TYPE+BINARYCONTENT_TYPE;
HTTP header for MIME content type used for raw binary data

BINARYCONTENT_TYPE_UPPER = 'APPLICATION/OCCKET-STREAM';
MIME content type used for raw binary data, in upper case

JSON compatible representation of a boolean value, i.e. 'false' and 'true'
- can be used e.g. in logs, or anything accepting a shortstring

CODEPAGE_LATIN1 = 819;
Latin-1 ISO/IEC 8859-1 Code Page

CODEPAGE_US = 1252;
US English Windows Code Page, i.e. WinAnsi standard character encoding

COMP_TEXT = 'Delphi';
The compiler family used

CPU_ARCH_TEXT = 'x86';
The CPU architecture used for compilation

CP_RAWBYTESTRING = 65535;
Internal Code Page for RawByteString undefined string

CP_SQLRAWBLOB = 65534;
Fake code page used to recognize TSQLRawBlob
- as returned e.g. by TTypeInfo.AnsiStringCodePage from mORMot.pas

CP_UTF16 = 1200;
Internal Code Page for UTF-16 Unicode encoding
- used e.g. for Delphi 2009+ UnicodeString=String type

djObject = djPointer;
TDynArrayKind alias for a TObject field hashing / comparison
djPointer = djCardinal;

TDynArrayKind alias for a pointer field hashing / comparison

DocVariantDataFake: TDocVariantData = ( VType: 1;
VOptions: [dvoReturnNullForUnknownProperty]);

Constant used e.g. by _Safe() overloaded functions
- will be in code section of the exe, so will be read-only by design
- would have Kind=dvUndefined and Count=0, so _Safe() would return a valid, but void document
- its VType is varNull, so would be viewed as a null variant
- dvoReturnNullForUnknownProperty is defined, so that U[]/I[]... methods won't raise any exception
about unexpected field name

DOUBLE_SAME = 1E-11;

A typical error allowed when working with double floating-point values
- 1E-12 is too small, and triggers sometimes some unexpected errors; FPC RTL uses 1E-4 so we are
paranoid enough

FILES_ALL = '*.*';

Operating-system dependent wildchar to match all files in a folder

GUID_NULL: TGUID = ();

A TGUID containing '{00000000-0000-0000-0000-000000000000}'

HASH_PO2 = 1 shl 18;

Defined for inlining bitwise division in TDynArrayHasher.HashTableIndex
- HashTableSize<=HASH_PO2 is expected to be a power of two (fast binary op); limit is set to 262,144
hash table slots (=1MB), for Capacity=131,072 items
- above this limit, a set of increasing primes is used; using a prime as hashtable modulo enhances its
distribution, especially for a weak hash function
- 64-bit CPU and FPC can efficiently compute a prime reduction using Lemire algorithm, so no power
of two is defined on those targets

HEADER_BEARER_UPPER = 'AUTHORIZATION: BEARER ';

HTTP header name for the authorization token, in upper case
- could be used e.g. with IdemPChar() to retrieve a JWT value
- will detect header computed e.g. by SynCrtSock.AuthorizationBearer()

HEADER_CONTENT_TYPE = 'Content-Type: ';

HTTP header name for the content type, as defined in the corresponding RFC

HEADER_CONTENT_TYPE_UPPER = 'CONTENT-TYPE: ';

HTTP header name for the content type, in upper case
- as defined in the corresponding RFC
- could be used e.g. with IdemPChar() to retrieve the Content-Type value

HEADER_REMOTEIP_UPPER = 'REMOTEIP: ';

HTTP header name for the client IP, in upper case
- as defined in our HTTP server classes
- could be used e.g. with IdemPChar() to retrieve the remote IP address

HTML_CONTENT_TYPE = 'text/html; charset=UTF-8';

MIME content type used for UTF-8 encoded HTML

HTML_CONTENT_TYPE_HEADER = HEADER_CONTENT_TYPE+HTML_CONTENT_TYPE;
HTTP header for MIME content type used for UTF-8 encoded HTML

JPEG_CONTENT_TYPE = 'image/jpeg';

MIME content type used for a JPEG picture

JSON_BASE64_MAGIC = $b0bfef;

UTF-8 encoded \uFFF0 special code to mark Base64 binary content in JSON
- Unicode special char U+FFF0 is UTF-8 encoded as EF BF 80 bytes
- as generated by BinToBase64WithMagic() functions, and expected by SQLParamContent() and ExtractInlineParameters() functions
- used e.g. when transmitting TDynArray.SaveTo() content

JSON_BASE64_MAGIC_QUOTE = ord('"')+cardinal(JSON_BASE64_MAGIC) shl 8;

"" + UTF-8 encoded \uFFF0 special code to mark Base64 binary in JSON

JSON_BASE64_MAGIC_QUOTE_VAR: cardinal = JSON_BASE64_MAGIC_QUOTE;

"" + UTF-8 encoded \uFFF0 special code to mark Base64 binary in JSON
- defined as a cardinal variable to be used as:
  AddNoJSONEscape(@JSON_BASE64_MAGIC_QUOTE_VAR,4);

JSON_CONTENT_TYPE = 'application/json; charset=UTF-8';

MIME content type used for JSON communication (as used by the Microsoft WCF framework and the YUI framework)

Used for DI-2.1.2 (page 2545).

JSON_CONTENT_TYPE_HEADER = HEADER_CONTENT_TYPE+JSON_CONTENT_TYPE;

HTTP header for MIME content type used for plain JSON

JSON_CONTENT_TYPE_HEADER_UPPER = HEADER_CONTENT_TYPE_UPPER+JSON_CONTENT_TYPE_UPPER;

HTTP header for MIME content type used for plain JSON, in upper case
- could be used e.g. with IdemPChar() to retrieve the Content-Type value

JSON_CONTENT_TYPE_UPPER = 'APPLICATION/JSON';

MIME content type used for plain JSON, in upper case
- could be used e.g. with IdemPChar() to retrieve the Content-Type value


The JavaScript-like values of non-number IEEE constants
- as recognized by FloatToShortNan, and used by TTextWriter.Add() when serializing such single/double/extended floating-point values

JSON_OPTIONS: array[Boolean] of TDocVariantOptions = (dvoReturnNullForUnknownProperty,dvoValueCopiedByReference);

Some convenient TDocVariant options, as JSON_OPTIONS[CopiedByReference]
- JSON_OPTIONS[false] is e.g. _Json() and _JsonFmt() functions default
- JSON_OPTIONS[true] are used e.g. by _JsonFast() and _JsonFastFmt() functions

JSON_OPTIONS_FAST = [dvoReturnNullForUnknownProperty,dvoValueCopiedByReference];

Same as JSON_OPTIONS[true], but can not be used as PDocVariantOptions

JSON_OPTIONS_FAST_EXTENDED: TDocVariantOptions = [dvoReturnNullForUnknownProperty,dvoValueCopiedByReference,dvoSerializeAsExtendedJson];
TDocVariant options to be used so that JSON serialization would use the unquoted JSON syntax for field names

- you could use it e.g. on a TSQLRecord variant published field to reduce the JSON escape process during storage in the database, by customizing your TSQLModel instance:

  ```pascal
  (aModel.Props[TSQLMyRecord][`VariantProp'] as TSQLPropInfoRTTIVariant).
  DocVariantOptions := JSON_OPTIONS_FAST_EXTENDED;
  ```

  or - in a cleaner way - by overriding TSQLRecord.InternalDefineModel():

  ```pascal
  class procedure TSQLMyRecord.InternalDefineModel(Props: TSQLRecordProperties);
  begin
    (Props.Fields.ByName(`VariantProp') as TSQLPropInfoRTTIVariant).
    DocVariantOptions := JSON_OPTIONS_FAST_EXTENDED;
  end;
  ```

or to set all variant fields at once:

```pascal
class procedure TSQLMyRecord.InternalDefineModel(Props: TSQLRecordProperties);
begin
  Props.SetVariantFieldsDocVariantOptions(JSON_OPTIONS_FAST_EXTENDED);
end;
```

- consider using JSON_OPTIONS_NAMEVALUE[true] for case-sensitive TSynNameValue-like storage, or JSON_OPTIONS_FAST_EXTENDEDINTERN if you expect RawUTF8 names and values interning

```pascal
JSON_OPTIONS_FAST_EXTENDEDINTERN: TDocVariantOptions =
[dvoReturnNullForUnknownProperty,dvoValueCopiedByReference,
 dvoSerializeAsExtendedJson,dvoJSONParseDoNotTryCustomVariants,
 dvoInternNames,dvoInternValues];
```

TDocVariant options for JSON serialization with efficient storage

- i.e. unquoted JSON syntax for field names and RawUTF8 interning
- may be used e.g. for efficient persistence of similar data
- consider using JSON_OPTIONS_FAST_EXTENDED if you don’t expect RawUTF8 names and values interning, or need BSON variants parsing

```pascal
JSON_OPTIONS_FAST.StrictJSON: TDocVariantOptions =
[dvoReturnNullForUnknownProperty,dvoValueCopiedByReference,
 dvoJSONParseDoNotTryCustomVariants];
```

TDocVariant options which may be used for plain JSON parsing

- this won’t recognize any extended syntax

```pascal
JSON_OPTIONS_NAMEVALUE: array[boolean] of TDocVariantOptions = ( 
[dvoReturnNullForUnknownProperty,dvoValueCopiedByReference,
 dvoNameCaseSensitive],
[dvoReturnNullForUnknownProperty,dvoValueCopiedByReference,
 dvoNameCaseSensitive,dvoSerializeAsExtendedJson]);
```

TDocVariant options to be used for case-sensitive TSynNameValue-like storage, with optional extended JSON syntax serialization

- consider using JSON_OPTIONS_FAST_EXTENDED for case-insensitive objects

```pascal
JSON_OPTIONS_NAMEVALUEINTERN: array[boolean] of TDocVariantOptions = ( 
[dvoReturnNullForUnknownProperty,dvoValueCopiedByReference,
 dvoNameCaseSensitive,dvoInternNames,dvoInternValues],
[dvoReturnNullForUnknownProperty,dvoValueCopiedByReference,
 dvoNameCaseSensitive,dvoInternNames,dvoInternValues, dvoSerializeAsExtendedJson]);
```

TDocVariant options to be used for case-sensitive TSynNameValue-like storage, RawUTF8 interning and optional extended JSON syntax serialization

- consider using JSON_OPTIONS_FAST_EXTENDED for case-insensitive objects, or JSON_OPTIONS_NAMEVALUE[] if you don’t expect names and values interning

```pascal
JSON_SQLDATE_MAGIC = $b1bfef;
```

UTF-8 encoded \\uFFF1 special code to mark ISO-8601 SQLDATE in JSON
- e.g. "\UFF12012-05-04" pattern
- Unicode special char U+FFF1 is UTF-8 encoded as EF BF B1 bytes
- as generated by DateToSQL/DateTimeToSQL/TimeLogToSQL functions, and expected by SQLParamContent() and ExtractInlineParameters() functions

```pascal
JSON_SQLDATE_MAGIC_QUOTE = ord("\"") + cardinal(JSON_SQLDATE_MAGIC) shl 8;
```

```
"\uff12" + UTF-8 encoded \ufff1 special code to mark ISO-8601 SQLDATE in JSON
```

```pascal
JSON_SQLDATE_MAGIC_QUOTE_VAR: cardinal = JSON_SQLDATE_MAGIC_QUOTE;
```

```
"\uff12" + UTF-8 encoded \ufff1 special code to mark ISO-8601 SQLDATE in JSON
```

- defined as a cardinal variable to be used as:
  ```pascal
  AddNoJSONEscape(@JSON_SQLDATE_MAGIC_QUOTE_VAR, 4);
  ```

LOGESCAPELEN = 200;

* Maximum size, in bytes, of a TLogEscape / LogEscape() buffer

MAXLOGSIZE = 1024*1024;

- Rotate local log file if reached this size (1MB by default)
- .log file will be save as .log.bak file
- a new .log file is created
- used by AppendToTextFile() and LogToTextFile() functions (not TSynLog)

MAX_SQLFIELDS = 64;

* Maximum number of fields in a database Table
- is included in SynCommons so that all DB-related work will be able to share the same low-level types and functions (e.g. TSQLFieldBits, TJSONWriter, TSynTableStatement, TSynTable, TSQLRecordProperties)
- default is 64, but can be set to any value (64, 128, 192 and 256 optimized) changing the source below or using MAX_SQLFIELDS_128, MAX_SQLFIELDS_192 or MAX_SQLFIELDS_256 conditional directives for your project
- this constant is used internally to optimize memory usage in the generated asm code, and statically allocate some arrays for better speed
- note that due to compiler restriction, 256 is the maximum value (this is the maximum number of items in a Delphi/FPC set)

```pascal
MAX_SQLFIELDS_INCLUDINGID = MAX_SQLFIELDS+1;
```

* Sometimes, the ID field is included in a bits set

ORDTYPE_SIZE: array[TOrdType] of byte = (1,1,2,2,4,4);

* Quick retrieve how many bytes an ordinal consist in


* Translate one operating system (and distribution) into a single character
- may be used internally e.g. for a HTTP User-Agent header, as with TFileVersion.UserAgent

```pascal
OS_LINUX = [osLinux, osArch .. osAndroid];
```

* For Android ... J = Java VM the operating systems items which actually are Linux distributions

```pascal
OS_TEXT = 'Win';
```

* The target Operating System used for compilation, as text

PLURAL_FORM: array[boolean] of RawUTF8 = (',','s');
Can be used to append to most English nouns to form a plural
- see also the Plural function

POINTERAND = 3;
Could be used to compute the bitmask of a pointer integer

POINTERBITS = 32;
Could be used to check all bits on a pointer

POINTERSHR = 2;
Could be used to compute the index in a pointer list from its position

ptPtrInt = ptInteger;
Map a PtrInt type to the TJSONCustomParserRTTIType set

ptPtrUInt = ptCardinal;
Map a PtrUInt type to the TJSONCustomParserRTTIType set

PT_COMPLEXTYPES = [ptArray, ptRecord, ptCustom, ptTimeLog];
Which TJSONCustomParserRTTIType types are not simple types
- ptTimeLog is complex, since could be also TCreateTime or TModTime

SYNLZTRIG: array[boolean] of integer = (100, maxInt);
CompressionSizeTrigger parameter SYNLZTRIG[true] will disable then SynLZCompress() compression

SYNOPSE_FRAMEWORK_FULLVERSION = SYNOPSE_FRAMEWORK_VERSION;
A text including the version and the main active conditional options
- useful for low-level debugging purpose

SYNOPSE_FRAMEWORK_VERSION = '1.18.6135';
For TEvent and TCriticalSection for TObjectList needed for TSynMapFile .mab format the corresponding version of the freeware Synopse framework
- includes a commit increasing number (generated by SourceCodeRep tool)
- a similar constant shall be defined in SynCrtSock.pas

TEXT_CONTENT_TYPE = 'text/plain; charset=UTF-8';
MIME content type used for plain UTF-8 text

TEXT_CONTENT_TYPE_HEADER = HEADER_CONTENT_TYPE+TEXT_CONTENT_TYPE;
HTTP header for MIME content type used for plain UTF-8 text

tkOrdinalTypes = [tkInteger, tkChar, tkWChar, tkEnumeration, tkSet, tkInt64 ];
Maps 1, 8, 16 and 32 and 64-bit ordinal in TTypeKind RTTI enumerate

tkRecordKinds = tkRecord;
Maps record or object in TTypeKind RTTI enumerate

tkRecordTypes = [tkRecord];
Maps record or object in TTypeKind RTTI enumerate

tkStringTypes = [tkLString, tkWString ];
Maps long string in TTypeKind RTTI enumerate

TwoDigitLookup: packed array[0..99] of array[1..2] of AnsiChar =
('00', '01', '02', '03', '04', '05', '06', '07', '08', '09',
'10', '11', '12', '13', '14', '15', '16', '17', '18', '19',
'20', '21', '22', '23', '24', '25', '26', '27', '28', '29',

...
Fast lookup table for converting any decimal number from 0 to 99 into their ASCII equivalence
- our enhanced SysUtils.pas (normal and LVCL) contains the same array

UNIXTIME_MINIMAL = 1481187020;
A contemporary, but elapsed, TUnixTime second-based value
- corresponds to Thu, 08 Dec 2016 08:50:20 GMT
- may be used to check for a valid just-generated Unix timestamp value

varNativeString = varString;
This variant type will map the current string type
- depending on the compiler version

varSynUnicode = varOleStr;
This variant type will map the current SynUnicode type
- depending on the compiler version

varWord64 = 21;
Unsigned 64bit integer variant type
- currently called varUInt64 in Delphi (not defined in older versions), and varQWord in FPC

WINDOWS_32 = [w2000, wXP, wServer2003, wServer2003_R2, wVista, wServer2008, wSeven, wServer2008_R2, wEight, wServer2012, wEightOne, wServer2012R2, wTen, wServer2016];
The recognized Windows versions which are 32-bit

The recognized Windows versions, as plain text
- defined even outside MSWINDOWS to allow process e.g. from monitoring tools

XMLUTF8_HEADER = '<?xml version="1.0" encoding="UTF-8"?>
Standard header for an UTF-8 encoded XML file

XMLUTF8_NAMESPACE = '<contents xmlns="http://www.w3.org/2001/XMLSchema-instance">
Standard namespace for a generic XML File

XML_CONTENT_TYPE = 'text/xml; charset=UTF-8';
MIME content type used for UTF-8 encoded XML

XML_CONTENT_TYPE_HEADER = HEADER_CONTENT_TYPE+XML_CONTENT_TYPE;
HTTP header for MIME content type used for UTF-8 encoded XML

_DALEN = SizeOf(PtrInt);
Cross-compiler negative offset to TDynArrayRec.high/length field
- to be used inlined e.g. as PDALen(PtrUInt(Values)-_DALEN)^{$ifdef FPC}+1{$endif}

_DAREFCNT = Sizeof(TRefCnt)+_DALEN;
Cross-compiler negative offset to TDynArrayRec.refCnt field
- to be used inlined e.g. as PRefCnt(_PtrUInt(Values)-_DAREFCNT)^

.semantic{
  _STRENCNT = SizeOf(TRefCnt)+_STRENCNT;
}

Cross-compiler negative offset to TStrRec.length field
- to be used inlined e.g. as PStrLen(p-_STRENCNT)^

_functions or procedures implemented in the SynCommons unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddArrayOfConst</td>
<td>Append one or several values to a local &quot;array of const&quot; variable</td>
<td>939</td>
</tr>
<tr>
<td>AddGUID</td>
<td>Append one TGUID item to a TGUID dynamic array</td>
<td>940</td>
</tr>
<tr>
<td>AddInt64</td>
<td>Add a 64-bit integer value at the end of a dynamic array</td>
<td>940</td>
</tr>
<tr>
<td>AddInt64</td>
<td>Add a 64-bit integer array at the end of a dynamic array</td>
<td>940</td>
</tr>
<tr>
<td>AddInt64</td>
<td>Add a 64-bit integer value at the end of a dynamic array of integers</td>
<td>940</td>
</tr>
<tr>
<td>AddInt64Once</td>
<td>If not already existing, add a 64-bit integer value to a dynamic array</td>
<td>940</td>
</tr>
<tr>
<td>AddInt64Sorted</td>
<td>If not already existing, add a 64-bit integer value to a sorted dynamic array</td>
<td>940</td>
</tr>
<tr>
<td>AddInteger</td>
<td>Add an integer value at the end of a dynamic array of integers</td>
<td>940</td>
</tr>
<tr>
<td>AddInteger</td>
<td>Add an integer array at the end of a dynamic array of integer</td>
<td>940</td>
</tr>
<tr>
<td>AddInteger</td>
<td>Add an integer value at the end of a dynamic array of integers</td>
<td>940</td>
</tr>
<tr>
<td>AddInteger</td>
<td>Add an integer value at the end of a dynamic array of integers</td>
<td>940</td>
</tr>
<tr>
<td>AddPrefixToCSV</td>
<td>Append some prefix to all CSV values</td>
<td>940</td>
</tr>
<tr>
<td>AddRawUTF8</td>
<td>True if Value was added successfully in Values[]</td>
<td>941</td>
</tr>
<tr>
<td>AddRawUTF8</td>
<td>Add the Value to Values[], with an external count variable, for performance</td>
<td>941</td>
</tr>
<tr>
<td>AddSortedInteger</td>
<td>Add an integer value in a sorted dynamic array of integers</td>
<td>941</td>
</tr>
<tr>
<td>AddSortedInteger</td>
<td>Add an integer value in a sorted dynamic array of integers</td>
<td>941</td>
</tr>
<tr>
<td>AddSortedRawUTF8</td>
<td>Add a RawUTF8 value in an alphabetically sorted dynamic array of RawUTF8</td>
<td>941</td>
</tr>
<tr>
<td>AddToCSV</td>
<td>Append a Value to a CSV string</td>
<td>941</td>
</tr>
<tr>
<td>AddWord</td>
<td>Add a 16-bit integer value at the end of a dynamic array of integers</td>
<td>941</td>
</tr>
<tr>
<td>AndMemory</td>
<td>Logical AND of two memory buffers</td>
<td>941</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Ansi7ToString</td>
<td>Convert any Ansi 7 bit encoded String into a generic VCL Text</td>
<td>942</td>
</tr>
<tr>
<td>Ansi7ToString</td>
<td>Convert any Ansi 7 bit encoded String into a generic VCL Text</td>
<td>942</td>
</tr>
<tr>
<td>Ansi7ToString</td>
<td>Convert any Ansi 7 bit encoded String into a generic VCL Text</td>
<td>942</td>
</tr>
<tr>
<td>AnsiCharToUTF8</td>
<td>Convert an AnsiChar buffer (of a given code page) into a UTF-8 string</td>
<td>942</td>
</tr>
<tr>
<td>AnsiIComp</td>
<td>Fast WinAnsi comparison using the NormToUpper[] array for all 8 bits values</td>
<td>942</td>
</tr>
<tr>
<td>AnsiICompW</td>
<td>Fast case-insensitive Unicode comparison</td>
<td>942</td>
</tr>
<tr>
<td>AnyAnsiToUTF8</td>
<td>Direct conversion of an AnsiString with an unknown code page into an UTF-8 encoded String</td>
<td>942</td>
</tr>
<tr>
<td>AnyAnsiToUTF8</td>
<td>Direct conversion of an AnsiString with an unknown code page into an UTF-8 encoded String</td>
<td>942</td>
</tr>
<tr>
<td>AnyScanExists</td>
<td>Fast search of a binary value position in a fixed-size array</td>
<td>942</td>
</tr>
<tr>
<td>AnyScanIndex</td>
<td>Fast search of a binary value position in a fixed-size array</td>
<td>942</td>
</tr>
<tr>
<td>AnyTextFileToRawUTF8</td>
<td>Get text file contents (even Unicode or UTF8) and convert it into an UTF-8 string according to any BOM marker at the beginning of the file</td>
<td>942</td>
</tr>
<tr>
<td>AnyTextFileToString</td>
<td>Get text file contents (even Unicode or UTF8) and convert it into a CharSet-compatible AnsiString (for Delphi 7) or an UnicodeString (for Delphi 2009 and up) according to any BOM marker at the beginning of the file</td>
<td>942</td>
</tr>
<tr>
<td>AnyTextFileToSynUnicode</td>
<td>Get text file contents (even Unicode or UTF8) and convert it into an Unicode string according to any BOM marker at the beginning of the file</td>
<td>943</td>
</tr>
<tr>
<td>Append999ToBuffer</td>
<td>Fast add text conversion of 0-999 integer value into a given buffer</td>
<td>943</td>
</tr>
<tr>
<td>AppendBuffersToRawUTF8</td>
<td>Fast add some characters to a RawUTF8 string</td>
<td>943</td>
</tr>
<tr>
<td>AppendBufferToRawUTF8</td>
<td>Fast add some characters to a RawUTF8 string</td>
<td>943</td>
</tr>
<tr>
<td>AppendCharToRawUTF8</td>
<td>Fast add one character to a RawUTF8 string</td>
<td>943</td>
</tr>
<tr>
<td>AppendCSVValues</td>
<td>Append some text lines with the supplied Values[]</td>
<td>943</td>
</tr>
<tr>
<td>AppendRawUTF8ToBuffer</td>
<td>Fast add some characters from a RawUTF8 string into a given buffer</td>
<td>943</td>
</tr>
<tr>
<td>AppendShortComma</td>
<td>Fast append some UTF-8 text into a shortstring, with an ending ','</td>
<td>943</td>
</tr>
<tr>
<td>AppendToFile</td>
<td>Log a message to a local text file</td>
<td>943</td>
</tr>
<tr>
<td>AppendUInt32ToBuffer</td>
<td>Fast add text conversion of a 32-bit unsigned integer value into a given buffer</td>
<td>943</td>
</tr>
<tr>
<td>AreUrlValid</td>
<td>Checks if the supplied UTF-8 text values don't need URI encoding</td>
<td>944</td>
</tr>
</tbody>
</table>
## Functions or procedures

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayOfConstValueAsText</td>
<td>Find a given name in name/value pairs, and returns the value as RawUTF8</td>
<td>944</td>
</tr>
<tr>
<td>AsciiToBaudot</td>
<td>Convert some ASCII-7 text into binary, using Emile Baudot code</td>
<td>944</td>
</tr>
<tr>
<td>AsciiToBaudot</td>
<td>Convert some ASCII-7 text into binary, using Emile Baudot code</td>
<td>944</td>
</tr>
<tr>
<td>Base64Decode</td>
<td>Direct low-level decoding of a Base64 encoded buffer</td>
<td>944</td>
</tr>
<tr>
<td>Base64MagicCheckAndDecode</td>
<td>Check and decode \uFFF0base64encodedbinary content into binary</td>
<td>944</td>
</tr>
<tr>
<td>Base64MagicCheckAndDecode</td>
<td>Check and decode \uFFF0base64encodedbinary content into binary</td>
<td>944</td>
</tr>
<tr>
<td>Base64MagicCheckAndDecode</td>
<td>Check and decode \uFFF0base64encodedbinary content into binary</td>
<td>944</td>
</tr>
<tr>
<td>Base64MagicDecode</td>
<td>Just a wrapper around Base64ToBin() for in-place decode of JSON_BASE64_MAGIC \uFFF0base64encodedbinary content into binary</td>
<td>945</td>
</tr>
<tr>
<td>Base64ToBin</td>
<td>Fast conversion from Base64 encoded text into binary data</td>
<td>945</td>
</tr>
<tr>
<td>Base64ToBin</td>
<td>Fast conversion from Base64 encoded text into binary data</td>
<td>945</td>
</tr>
<tr>
<td>Base64ToBin</td>
<td>Fast conversion from Base64 encoded text into binary data</td>
<td>945</td>
</tr>
<tr>
<td>Base64ToBin</td>
<td>Fast conversion from Base64 encoded text into binary data</td>
<td>945</td>
</tr>
<tr>
<td>Base64ToBin</td>
<td>Fast conversion from Base64 encoded text into binary data</td>
<td>945</td>
</tr>
<tr>
<td>Base64ToBin</td>
<td>Fast conversion from Base64 encoded text into binary data</td>
<td>945</td>
</tr>
<tr>
<td>Base64ToBin</td>
<td>Fast conversion from Base64 encoded text into binary data</td>
<td>945</td>
</tr>
<tr>
<td>Base64ToBinLength</td>
<td>Retrieve the expected undecoded length of a Base64 encoded buffer</td>
<td>945</td>
</tr>
<tr>
<td>Base64ToBinLengthSafe</td>
<td>Retrieve the expected undecoded length of a Base64 encoded buffer</td>
<td>945</td>
</tr>
<tr>
<td>Base64ToBinSafe</td>
<td>Fast conversion from Base64 encoded text into binary data</td>
<td>946</td>
</tr>
<tr>
<td>Base64ToBinSafe</td>
<td>Fast conversion from Base64 encoded text into binary data</td>
<td>946</td>
</tr>
<tr>
<td>Base64ToBinSafe</td>
<td>Fast conversion from Base64 encoded text into binary data</td>
<td>946</td>
</tr>
<tr>
<td>Base64ToURI</td>
<td>Conversion from any Base64 encoded value into URI-compatible encoded text</td>
<td>946</td>
</tr>
<tr>
<td>Base64uriDecode</td>
<td>Direct low-level decoding of a Base64-URI encoded buffer</td>
<td>946</td>
</tr>
<tr>
<td>Base64uriEncode</td>
<td>Low-level conversion from a binary buffer into Base64-like URI-compatible encoded text</td>
<td>946</td>
</tr>
<tr>
<td>Base64uriToBin</td>
<td>Fast conversion from Base64-URI encoded text into binary data</td>
<td>947</td>
</tr>
<tr>
<td>Base64uriToBin</td>
<td>Fast conversion from Base64-URI encoded text into binary data</td>
<td>947</td>
</tr>
<tr>
<td>Base64uriToBin</td>
<td>Fast conversion from Base64-URI encoded text into binary data</td>
<td>947</td>
</tr>
</tbody>
</table>
### Functions or procedures

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base64uriToBin</td>
<td>Fast conversion from Base64-URI encoded text into binary data</td>
<td>947</td>
</tr>
<tr>
<td>Base64uriToBin</td>
<td>Fast conversion from Base64-URI encoded text into binary data</td>
<td>947</td>
</tr>
<tr>
<td>Base64uriToBin</td>
<td>Fast conversion from Base64-URI encoded text into binary data</td>
<td>947</td>
</tr>
<tr>
<td>Base64uriToBinLength</td>
<td>Retrieve the expected undecoded length of a Base64-URI encoded buffer</td>
<td>947</td>
</tr>
<tr>
<td>BaudotToAscii</td>
<td>Convert some Baudot code binary, into ASCII-7 text</td>
<td>947</td>
</tr>
<tr>
<td>BaudotToAscii</td>
<td>Convert some Baudot code binary, into ASCII-7 text</td>
<td>947</td>
</tr>
<tr>
<td>BinToBase64</td>
<td>Fast conversion from binary data into Base64 encoded UTF-8 text</td>
<td>947</td>
</tr>
<tr>
<td>BinToBase64</td>
<td>Fast conversion from binary data into Base64 encoded UTF-8 text</td>
<td>947</td>
</tr>
<tr>
<td>BinToBase64</td>
<td>Fast conversion from binary data into prefixed/suffixed Base64 encoded UTF-8 text</td>
<td>947</td>
</tr>
<tr>
<td>BinToBase64Length</td>
<td>Retrieve the expected encoded length after Base64 process</td>
<td>947</td>
</tr>
<tr>
<td>BinToBase64Short</td>
<td>Fast conversion from a small binary data into Base64 encoded UTF-8 text</td>
<td>947</td>
</tr>
<tr>
<td>BinToBase64Short</td>
<td>Fast conversion from a small binary data into Base64 encoded UTF-8 text</td>
<td>947</td>
</tr>
<tr>
<td>BinToBase64uri</td>
<td>Fast conversion from binary data into Base64-like URI-compatible encoded text</td>
<td>948</td>
</tr>
<tr>
<td>BinToBase64uri</td>
<td>Fast conversion from a binary buffer into Base64-like URI-compatible encoded text</td>
<td>948</td>
</tr>
<tr>
<td>BinToBase64uriLength</td>
<td>Retrieve the expected encoded length after Base64-URI process</td>
<td>948</td>
</tr>
<tr>
<td>BinToBase64uriShort</td>
<td>Fast conversion from a binary buffer into Base64-like URI-compatible encoded shortstring</td>
<td>948</td>
</tr>
<tr>
<td>BinToBase64WithMagic</td>
<td>Fast conversion from binary data into Base64 encoded UTF-8 text with JSON_BASE64_MAGIC prefix (UTF-8 encoded \uFFF0 special code)</td>
<td>948</td>
</tr>
<tr>
<td>BinToBase64WithMagic</td>
<td>Fast conversion from binary data into Base64 encoded UTF-8 text with JSON_BASE64_MAGIC prefix (UTF-8 encoded \uFFF0 special code)</td>
<td>948</td>
</tr>
<tr>
<td>BinToHex</td>
<td>Fast conversion from binary data into hexa chars</td>
<td>948</td>
</tr>
<tr>
<td>BinToHex</td>
<td>Fast conversion from binary data into hexa chars</td>
<td>948</td>
</tr>
<tr>
<td>BinToHex</td>
<td>Fast conversion from binary data into hexa chars</td>
<td>948</td>
</tr>
<tr>
<td>BinToHexDisplay</td>
<td>Fast conversion from binary data into hexa chars, ready to be displayed</td>
<td>948</td>
</tr>
<tr>
<td>BinToHexDisplay</td>
<td>Fast conversion from binary data into hexa chars, ready to be displayed</td>
<td>948</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>BinToHexDisplayFile</td>
<td>Fast conversion from binary data into hexa lowercase chars, ready to be used as a convenient TFileName prefix</td>
<td>949</td>
</tr>
<tr>
<td>BinToHexDisplayLower</td>
<td>Fast conversion from binary data into lowercase hexa chars</td>
<td>949</td>
</tr>
<tr>
<td>BinToHexDisplayLower</td>
<td>Fast conversion from binary data into lowercase hexa chars</td>
<td>949</td>
</tr>
<tr>
<td>BinToHexDisplayLowerShort</td>
<td>Fast conversion from up to 127 bytes of binary data into lowercase hexa chars</td>
<td>949</td>
</tr>
<tr>
<td>BinToHexDisplayLowerShort16</td>
<td>Fast conversion from up to 64-bit of binary data into lowercase hexa chars</td>
<td>949</td>
</tr>
<tr>
<td>BinToHexDisplayLowerVariant</td>
<td>Fast conversion of a binary buffer into hexa chars, as a variant string</td>
<td>949</td>
</tr>
<tr>
<td>BinToHexLower</td>
<td>Fast conversion from binary data into lowercase hexa chars</td>
<td>949</td>
</tr>
<tr>
<td>BinToHexLower</td>
<td>Fast conversion from binary data into lowercase hexa chars</td>
<td>949</td>
</tr>
<tr>
<td>BinToHexLower</td>
<td>Fast conversion from binary data into lowercase hexa chars</td>
<td>949</td>
</tr>
<tr>
<td>BinToHexLower</td>
<td>Fast conversion from binary data into lowercase hexa chars</td>
<td>949</td>
</tr>
<tr>
<td>BinToSource</td>
<td>Generate some pascal source code holding some data binary as constant</td>
<td>950</td>
</tr>
<tr>
<td>BinToSource</td>
<td>Generate some pascal source code holding some data binary as constant</td>
<td>950</td>
</tr>
<tr>
<td>bswap32</td>
<td>Convert the endianness of a given unsigned 32-bit integer into BigEndian</td>
<td>950</td>
</tr>
<tr>
<td>bswap64</td>
<td>Convert the endianness of a given unsigned 64-bit integer into BigEndian</td>
<td>950</td>
</tr>
<tr>
<td>bswap64array</td>
<td>Convert the endianness of an array of unsigned 64-bit integer into BigEndian</td>
<td>950</td>
</tr>
<tr>
<td>BufferLineLength</td>
<td>Compute the line length from a size-delimited source array of chars</td>
<td>950</td>
</tr>
<tr>
<td>ByteScanIndex</td>
<td>Fast search of an unsigned Byte value position in a Byte array</td>
<td>950</td>
</tr>
<tr>
<td>BytesToRawByteString</td>
<td>Creates a RawByteString memory buffer from a TBytes content</td>
<td>950</td>
</tr>
<tr>
<td>ByteToHex</td>
<td>Append one byte as hexadecimal char pairs, into a text buffer</td>
<td>950</td>
</tr>
<tr>
<td>CamelCase</td>
<td>Convert a string into an human-friendly CamelCase identifier</td>
<td>950</td>
</tr>
<tr>
<td>CamelCase</td>
<td>Convert a string into an human-friendly CamelCase identifier</td>
<td>950</td>
</tr>
<tr>
<td>CardinalToHex</td>
<td>Fast conversion from a Cardinal value into hexa chars, ready to be displayed</td>
<td>951</td>
</tr>
<tr>
<td>CardinalToHexLower</td>
<td>Fast conversion from a Cardinal value into hexa chars, ready to be displayed</td>
<td>951</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>CardinalToHexShort</td>
<td>Fast conversion from a Cardinal value into hexa chars, ready to be displayed</td>
<td>951</td>
</tr>
<tr>
<td>Char2ToByte</td>
<td>Fast conversion of 2 digit characters into a 0..99 value</td>
<td>951</td>
</tr>
<tr>
<td>Char3ToWord</td>
<td>Fast conversion of 3 digit characters into a 0..9999 value</td>
<td>951</td>
</tr>
<tr>
<td>Char4ToWord</td>
<td>Fast conversion of 4 digit characters into a 0..9999 value</td>
<td>951</td>
</tr>
<tr>
<td>Chars3ToInt18</td>
<td>Revert the value as encoded by TTextWriter.AddInt18ToChars3() or Int18ToChars3()</td>
<td>951</td>
</tr>
<tr>
<td>CharSetToCodePage</td>
<td>Convert a char set to a code page</td>
<td>951</td>
</tr>
<tr>
<td>ClassNameShort</td>
<td>Just a wrapper around vmtClassName to avoid a string conversion</td>
<td>951</td>
</tr>
<tr>
<td>CompareCardinal</td>
<td>A comparison function for sorting 32-bit unsigned integer values</td>
<td>951</td>
</tr>
<tr>
<td>CompareFloat</td>
<td>A comparison function for sorting IEEE 754 double precision values</td>
<td>951</td>
</tr>
<tr>
<td>CompareInt64</td>
<td>A comparison function for sorting 64-bit signed integer values</td>
<td>951</td>
</tr>
<tr>
<td>CompareInteger</td>
<td>A comparison function for sorting 32-bit signed integer values</td>
<td>951</td>
</tr>
<tr>
<td>CompareMem</td>
<td>Our fast version of CompareMem() with optimized asm for x86 and tune pascal</td>
<td>952</td>
</tr>
<tr>
<td>CompareMemSmall</td>
<td>A CompareMem()-like function designed for small (a few bytes) content</td>
<td>952</td>
</tr>
<tr>
<td>CompareQWord</td>
<td>A comparison function for sorting 64-bit unsigned integer values</td>
<td>952</td>
</tr>
<tr>
<td>CompressSynLZ</td>
<td>Compress a data content using the SynLZ algorithm</td>
<td>952</td>
</tr>
<tr>
<td>ContainsUTF8</td>
<td>Return true if up^ is contained inside the UTF-8 buffer p^</td>
<td>952</td>
</tr>
<tr>
<td>ConvertCaseUTF8</td>
<td>Fast conversion of the supplied text into 8 bit case sensitivity</td>
<td>952</td>
</tr>
<tr>
<td>CopyAndSortInt64</td>
<td>Copy an integer array, then sort it, low values first</td>
<td>952</td>
</tr>
<tr>
<td>CopyAndSortInteger</td>
<td>Copy an integer array, then sort it, low values first</td>
<td>952</td>
</tr>
<tr>
<td>CopyFile</td>
<td>Copy one file to another, similar to the Windows API</td>
<td>952</td>
</tr>
<tr>
<td>CopyInt64</td>
<td>Create a new 64-bit integer dynamic array with the values from another one</td>
<td>952</td>
</tr>
<tr>
<td>CopyInteger</td>
<td>Create a new 32-bit integer dynamic array with the values from another one</td>
<td>952</td>
</tr>
<tr>
<td>crc128c</td>
<td>Compute a 128-bit checksum on the supplied buffer, cascading two crc32c</td>
<td>952</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>crc16</td>
<td>Compute CRC16-CCITT checksum on the supplied buffer</td>
<td>953</td>
</tr>
<tr>
<td>crc256c</td>
<td>Compute a 256-bit checksum on the supplied buffer using crc32c</td>
<td>953</td>
</tr>
<tr>
<td>crc32cfast</td>
<td>Compute CRC32C checksum on the supplied buffer on processor-neutral code</td>
<td>953</td>
</tr>
<tr>
<td>crc32cinline</td>
<td>Compute CRC32C checksum on the supplied buffer using inlined code</td>
<td>953</td>
</tr>
<tr>
<td>crc32csse42</td>
<td>Compute CRC32C checksum on the supplied buffer using SSE 4.2</td>
<td>953</td>
</tr>
<tr>
<td>crc32cUTF8ToHex</td>
<td>Compute the hexadecimal representation of the crc32 checksum of a given text</td>
<td>953</td>
</tr>
<tr>
<td>crc512c</td>
<td>Compute a 512-bit checksum on the supplied buffer using crc32c</td>
<td>953</td>
</tr>
<tr>
<td>crc63c</td>
<td>Compute CRC63C checksum on the supplied buffer, cascading two crc32c</td>
<td>953</td>
</tr>
<tr>
<td>crc64c</td>
<td>Compute CRC64C checksum on the supplied buffer, cascading two crc32c</td>
<td>954</td>
</tr>
<tr>
<td>crcblockNoSSE42</td>
<td>Computation of our 128-bit CRC of a 128-bit binary buffer without SSE4.2</td>
<td>954</td>
</tr>
<tr>
<td>crcblocksfast</td>
<td>Compute a proprietary 128-bit CRC of 128-bit binary buffers</td>
<td>954</td>
</tr>
<tr>
<td>CreateInternalWindow</td>
<td>This function can be used to create a GDI compatible window, able to receive Windows Messages for fast local communication</td>
<td>954</td>
</tr>
<tr>
<td>CSVEncode</td>
<td>Encode name/value pairs into CSV/INI raw format</td>
<td>954</td>
</tr>
<tr>
<td>CSVOfValue</td>
<td>Return a CSV list of the iterated same value</td>
<td>954</td>
</tr>
<tr>
<td>CSVToInt64DynArray</td>
<td>Append the strings in the specified CSV text into a dynamic array of integer</td>
<td>954</td>
</tr>
<tr>
<td>CSVToInt64DynArray</td>
<td>Convert the strings in the specified CSV text into a dynamic array of integer</td>
<td>954</td>
</tr>
<tr>
<td>CSVToIntegerDynArray</td>
<td>Append the strings in the specified CSV text into a dynamic array of integer</td>
<td>954</td>
</tr>
<tr>
<td>CSVToRawUTF8DynArray</td>
<td>Add the strings in the specified CSV text into a dynamic array of UTF-8 strings</td>
<td>954</td>
</tr>
<tr>
<td>CSVToRawUTF8DynArray</td>
<td>Add the strings in the specified CSV text into a dynamic array of UTF-8 strings</td>
<td>954</td>
</tr>
<tr>
<td>Curr64ToPChar</td>
<td>Convert an INTEGER Curr64 (value*10000) into a string</td>
<td>955</td>
</tr>
<tr>
<td>Curr64ToStr</td>
<td>Convert an INTEGER Curr64 (value*10000) into a string</td>
<td>955</td>
</tr>
<tr>
<td>Curr64ToStr</td>
<td>Convert an INTEGER Curr64 (value*10000) into a string</td>
<td>955</td>
</tr>
</tbody>
</table>
## Functions or procedures

<table>
<thead>
<tr>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert a currency value from its Int64 binary representation into its numerical text equivalency</td>
<td>955</td>
</tr>
<tr>
<td>Convert a currency value into a string</td>
<td>955</td>
</tr>
<tr>
<td>Convert some date/time to the ISO 8601 text layout, including milliseconds</td>
<td>955</td>
</tr>
<tr>
<td>Convert some date/time to the ISO 8601 text layout, including milliseconds</td>
<td>955</td>
</tr>
<tr>
<td>Convert some TDateTime to a small text layout, perfect e.g. for naming a local file</td>
<td>955</td>
</tr>
<tr>
<td>Delphi 2007 is buggy as hell convert some TDateTime to a small text layout, perfect e.g. for naming a local file</td>
<td>955</td>
</tr>
<tr>
<td>Convert some date/time to the &quot;HTTP-date&quot; format as defined by RFC 7231</td>
<td>956</td>
</tr>
<tr>
<td>Wrapper calling global i18nDateTimeText() callback if set, or returning ISO-8601 standard layout on default</td>
<td>956</td>
</tr>
<tr>
<td>Basic Date/Time conversion into ISO-8601</td>
<td>956</td>
</tr>
<tr>
<td>Basic Date/Time conversion into ISO-8601</td>
<td>956</td>
</tr>
<tr>
<td>Write a TDateTime value, expanded as Iso-8601 encoded text into P^ Ansi buffer</td>
<td>956</td>
</tr>
<tr>
<td>Write a TDateTime into strict ISO-8601 date and/or time text</td>
<td>956</td>
</tr>
<tr>
<td>Write a TDateTime into strict ISO-8601 date and/or time text</td>
<td>957</td>
</tr>
<tr>
<td>Write a TDateTime into strict ISO-8601 date and/or time text</td>
<td>957</td>
</tr>
<tr>
<td>Convert a TDateTime into a millisecond-based c-encoded time (from Unix epoch 1/1/1970)</td>
<td>957</td>
</tr>
<tr>
<td>Convert a TDateTime into a second-based c-encoded time</td>
<td>957</td>
</tr>
<tr>
<td>Basic Date conversion into ISO-8601</td>
<td>957</td>
</tr>
<tr>
<td>Basic Date conversion into ISO-8601</td>
<td>957</td>
</tr>
<tr>
<td>Write a Date/Time to P^ Ansi buffer</td>
<td>957</td>
</tr>
<tr>
<td>Write a Date to P^ Ansi buffer</td>
<td>957</td>
</tr>
<tr>
<td>Convert a date into 'YYYY-MM-DD' date format</td>
<td>957</td>
</tr>
<tr>
<td>Basic Date period conversion into ISO-8601</td>
<td>958</td>
</tr>
<tr>
<td>Sort and remove any 64-bit duplicated integer from Values[]</td>
<td>958</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>DeduplicateInt64</td>
<td>Sort and remove any 64-bit duplicated integer from Values[]</td>
</tr>
<tr>
<td>DeduplicateInt64Sorted</td>
<td>Low-level function called by DeduplicateInt64()</td>
</tr>
<tr>
<td>DeduplicateInteger</td>
<td>Sort and remove any 32-bit duplicated integer from Values[]</td>
</tr>
<tr>
<td>DeduplicateIntegerSorted</td>
<td>Low-level function called by DeduplicateInteger()</td>
</tr>
<tr>
<td>DeleteCriticalSectionIfNeeded</td>
<td>On need finalization of a mutex</td>
</tr>
<tr>
<td>DeleteInt64</td>
<td>Delete any 64-bit integer in Values[]</td>
</tr>
<tr>
<td>DeleteInt64</td>
<td>Delete any 64-bit integer in Values[]</td>
</tr>
<tr>
<td>DeleteInteger</td>
<td>Delete any 32-bit integer in Values[]</td>
</tr>
<tr>
<td>DeleteInteger</td>
<td>Delete any 32-bit integer in Values[]</td>
</tr>
<tr>
<td>DeleteRawUTF8</td>
<td>Delete a RawUTF8 item in a dynamic array of RawUTF8</td>
</tr>
<tr>
<td>DeleteRawUTF8</td>
<td>Delete a RawUTF8 item in a dynamic array of RawUTF8;</td>
</tr>
<tr>
<td>DeleteSection</td>
<td>Delete a whole [Section]</td>
</tr>
<tr>
<td>DeleteSection</td>
<td>Delete a whole [Section]</td>
</tr>
<tr>
<td>DeleteWord</td>
<td>Delete any 16-bit integer in Values[]</td>
</tr>
<tr>
<td>DirectoryDelete</td>
<td>Delete the content of a specified directory</td>
</tr>
<tr>
<td>DirectoryDeleteOlderFiles</td>
<td>Delete the files older than a given age in a specified directory</td>
</tr>
<tr>
<td>Div100</td>
<td>Simple wrapper to efficiently compute both division and modulo per 100</td>
</tr>
<tr>
<td>DocVariantData</td>
<td>Direct access to a TDocVariantData from a given variant instance</td>
</tr>
<tr>
<td>DoubleToJSON</td>
<td>Convert a 64-bit floating-point value to its JSON text equivalency</td>
</tr>
<tr>
<td>DoubleToShort</td>
<td>Convert a 64-bit floating-point value to its numerical text equivalency</td>
</tr>
<tr>
<td>DoubleToShortNoExp</td>
<td>Convert a 64-bit floating-point value to its numerical text equivalency without scientific notation</td>
</tr>
<tr>
<td>DoubleToStr</td>
<td>Convert a 64-bit floating-point value to its numerical text equivalency</td>
</tr>
<tr>
<td>DoubleToString</td>
<td>Convert a floating-point value to its numerical text equivalency</td>
</tr>
<tr>
<td>DynArray</td>
<td>Initialize the structure with a one-dimension dynamic array</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>DynArrayBlobSaveJSON</td>
<td>Serialize a dynamic array content, supplied as raw binary buffer, as JSON</td>
</tr>
<tr>
<td>DynArrayCopy</td>
<td>Copy a dynamic array content from source to Dest</td>
</tr>
<tr>
<td>DynArrayElementTypeNa-</td>
<td>Compute a dynamic array element information</td>
</tr>
<tr>
<td>me</td>
<td>Compare two dynamic arrays by calling TDynArray.Equals</td>
</tr>
<tr>
<td>DynArrayItemTypeInfoS-</td>
<td>Was dynamic array item after RegisterCustomJSONSerializerFromTextBinaryType()</td>
</tr>
<tr>
<td>mpleBinary</td>
<td>Trim ending 'DynArray' or 's' chars from a dynamic array type name</td>
</tr>
<tr>
<td>DynArrayLoad</td>
<td>Fill a dynamic array content from a binary serialization as saved by DynArraySave() / TDynArray.Save()</td>
</tr>
<tr>
<td>DynArrayLoadJSON</td>
<td>Fill a dynamic array content from a JSON serialization as saved by TTextWriter.AddDynArrayJSON</td>
</tr>
<tr>
<td>DynArrayLoadJSON</td>
<td>Fill a dynamic array content from a JSON serialization as saved by TTextWriter.AddDynArrayJSON, which won't be modified</td>
</tr>
<tr>
<td>DynArraySave</td>
<td>Serialize a dynamic array content as binary, ready to be loaded by DynArrayLoad() / TDynArray.Load()</td>
</tr>
<tr>
<td>DynArraySaveJSON</td>
<td>Serialize a dynamic array content as JSON</td>
</tr>
<tr>
<td>DynArraySortIndexed</td>
<td>Sort any dynamic array, via an external array of indexes</td>
</tr>
<tr>
<td>DynArrayTypeInfoToRec-</td>
<td>Retrieve the item type information of a dynamic array low-level RTTI</td>
</tr>
<tr>
<td>ordInfo</td>
<td></td>
</tr>
<tr>
<td>Elapsed</td>
<td>Check if the current timestamp, in ms, matched a given period</td>
</tr>
<tr>
<td>EndWith</td>
<td>Check matching ending of p^ in upText</td>
</tr>
<tr>
<td>EndWithArray</td>
<td>Returns the index of a matching ending of p^ in upArray[]</td>
</tr>
<tr>
<td>EnsureDirectoryExists</td>
<td>Creates a directory if not already existing</td>
</tr>
<tr>
<td>EscapeBuffer</td>
<td>Fast conversion from binary data to escaped text</td>
</tr>
<tr>
<td>EscapeToShort</td>
<td>Fill a shortstring with the (hexadecimal) chars of the input text/binary</td>
</tr>
<tr>
<td>EscapeToShort</td>
<td>Fill a shortstring with the (hexadecimal) chars of the input text/binary</td>
</tr>
<tr>
<td>EventEquals</td>
<td>Compare two TMethod instances</td>
</tr>
<tr>
<td>ExcludeInt64</td>
<td>Remove some 64-bit integer from Values[]</td>
</tr>
<tr>
<td>ExcludeInteger</td>
<td>Remove some 32-bit integer from Values[]</td>
</tr>
<tr>
<td>ExistsIniName</td>
<td>Return TRUE if Value of UpperName does exist in P, till end of current section</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ExistsIniNameValue</td>
<td>Return TRUE if one of the Value of UpperName exists in P, till end of current section</td>
</tr>
<tr>
<td>ExtendedToJSON</td>
<td>Convert a floating-point value to its JSON text equivalency</td>
</tr>
<tr>
<td>ExtendedToShort</td>
<td>Convert a floating-point value to its numerical text equivalency</td>
</tr>
<tr>
<td>ExtendedToShortNoExp</td>
<td>Convert a floating-point value to its numerical text equivalency without scientification notation</td>
</tr>
<tr>
<td>ExtendedToStr</td>
<td>Convert a floating-point value to its numerical text equivalency</td>
</tr>
<tr>
<td>FastDynArrayClear</td>
<td>Low-level finalization of a dynamic array of variants</td>
</tr>
<tr>
<td>FastFindIndexedPUTF8Char</td>
<td>Retrieve the index of a PUTF8Char in a PUTF8Char array via a sort indexed</td>
</tr>
<tr>
<td>FastFindInt64Sorted</td>
<td>Fast O(log(n)) binary search of a 64-bit signed integer value in a sorted array</td>
</tr>
<tr>
<td>FastFindIntegerSorted</td>
<td>Fast O(log(n)) binary search of an integer value in a sorted integer array</td>
</tr>
<tr>
<td>FastFindIntegerSorted</td>
<td>Fast O(log(n)) binary search of an integer value in a sorted integer array</td>
</tr>
<tr>
<td>FastFindPointerSorted</td>
<td>Fast O(log(n)) binary search of a Pointer value in a sorted array</td>
</tr>
<tr>
<td>FastFindPtrIntSorted</td>
<td>Fast O(log(n)) binary search of a PtrInt value in a sorted array</td>
</tr>
<tr>
<td>FastFindPUTF8CharSorted</td>
<td>Retrieve the index where is located a PUTF8Char in a sorted PUTF8Char array</td>
</tr>
<tr>
<td>FastFindPUTF8CharSorted</td>
<td>Retrieve the index where is located a PUTF8Char in a sorted PUTF8Char array</td>
</tr>
<tr>
<td>FastFindQWordSorted</td>
<td>Fast O(log(n)) binary search of a 64-bit unsigned integer value in a sorted array</td>
</tr>
<tr>
<td>FastFindUpperPUTF8CharSorted</td>
<td>Retrieve the index where is located a PUTF8Char in a sorted uppercase PUTF8Char array</td>
</tr>
<tr>
<td>FastFindWordSorted</td>
<td>Fast O(log(n)) binary search of a 16 bit unsigned integer value in a sorted array</td>
</tr>
<tr>
<td>FastLocateIntegerSorted</td>
<td>Retrieve the index where to insert an integer value in a sorted integer array</td>
</tr>
<tr>
<td>FastLocatePUTF8CharSorted</td>
<td>Retrieve the index where to insert a PUTF8Char in a sorted PUTF8Char array</td>
</tr>
<tr>
<td>FastLocatePUTF8CharSorted</td>
<td>Retrieve the index where to insert a PUTF8Char in a sorted PUTF8Char array</td>
</tr>
<tr>
<td>FastLocateWordSorted</td>
<td>Retrieve the index where to insert a word value in a sorted word array</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FastSetString</td>
<td>Equivalence to SetString(s,nil,len) function</td>
</tr>
<tr>
<td>FastSetStringCP</td>
<td>Equivalence to SetString(s,nil,len) function with a specific code page</td>
</tr>
<tr>
<td>FileAgeToDateTine</td>
<td>Get a file date and time, from its name</td>
</tr>
<tr>
<td>FileFromStream</td>
<td>Create a File from a string content</td>
</tr>
<tr>
<td>FileInfoByHandle</td>
<td>Get low-level file information, in a cross-platform way</td>
</tr>
<tr>
<td>FileIsSynLZ</td>
<td>Returns TRUE if the supplied file name is a SynLZ compressed file,</td>
</tr>
<tr>
<td></td>
<td>matching the Magic number as supplied to FileSynLZ() function</td>
</tr>
<tr>
<td>FileOpenSequentialRead</td>
<td>Overloaded function optimized for one pass file reading</td>
</tr>
<tr>
<td>FileSeek64</td>
<td>FileSeek() overloaded function, working with huge files</td>
</tr>
<tr>
<td>FileSetDateFrom</td>
<td>Copy the date of one file to another</td>
</tr>
<tr>
<td>FileSize</td>
<td>Get a file size, from its handle</td>
</tr>
<tr>
<td>FileSize</td>
<td>Get a file size, from its name</td>
</tr>
<tr>
<td>FileStreamSequentialRead</td>
<td>Returns a TFileStream optimized for one pass file reading</td>
</tr>
<tr>
<td>FileSynLZ</td>
<td>Compress a file content using the SynLZ algorithm</td>
</tr>
<tr>
<td>FileTimeToInt64</td>
<td>Low-level wrapper to get the 64-bit value from a TFileTime</td>
</tr>
<tr>
<td>FileTimeToUnixMSTime</td>
<td>Low-level conversion of a Windows 64-bit TFileTime into a Unix time ms stamp</td>
</tr>
<tr>
<td>FileTimeToUnixTime</td>
<td>Low-level conversion of a Windows 64-bit TFileTime into a Unix time seconds stamp</td>
</tr>
<tr>
<td>FileUnSynLZ</td>
<td>Uncompress a file previously compressed via FileSynLZ()</td>
</tr>
<tr>
<td>FillIncreasing</td>
<td>Fill some values with i,i+1,i+2...i+Count-1</td>
</tr>
<tr>
<td>FillRandom</td>
<td>Fill some memory buffer with random values</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fill a GUID with 0</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fill all entries of a supplied array of RawUTF8 with &quot; &quot;</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fill all entries of a supplied array of 32-bit integers with 0</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fill all 20 bytes of this 160-bit buffer with zero</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fill all 16 bytes of this 128-bit buffer with zero</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fill all bytes of the value's memory buffer with zeros, i.e. 'toto' -&gt;</td>
</tr>
<tr>
<td></td>
<td>#0#0#0#0#0#0#0#0</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fill all 64 bytes of this 512-bit buffer with zero</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fill all entries of a supplied array of 64-bit integers with 0</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fill all 32 bytes of this 384-bit buffer with zero</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fill all 32 bytes of this 256-bit buffer with zero</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fill all bytes of this UTF-8 string with zeros, i.e. 'toto' -&gt; #0#0#0#0</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fill all bytes of this memory buffer with zeros, i.e. 'toto' -&gt; #0#0#0#0</td>
</tr>
<tr>
<td>FindAnsi</td>
<td>Return true if UpperValue (Ansi) is contained in A^ (Ansi)</td>
</tr>
<tr>
<td>FindCSVIndex</td>
<td>Return the index of a Value in a CSV string</td>
</tr>
<tr>
<td>FindFiles</td>
<td>Search for matching file names</td>
</tr>
<tr>
<td>FindFilesDynArrayToFileNames</td>
<td>Convert a result list, as returned by FindFiles(), into an array of Files[].Name</td>
</tr>
<tr>
<td>FindIniEntry</td>
<td>Find a Name= Value in a [Section] of a INI RawUTF8 Content</td>
</tr>
<tr>
<td>FindIniEntryFile</td>
<td>Find a Name= Value in a [Section] of a .INI file</td>
</tr>
<tr>
<td>FindIniEntryInteger</td>
<td>Find a Name= numeric Value in a [Section] of a INI RawUTF8 Content and return it as an integer, or 0 if not found</td>
</tr>
<tr>
<td>FindIniNameValue</td>
<td>Find the Value of UpperName in P, till end of current section</td>
</tr>
<tr>
<td>FindIniNameValueInteger</td>
<td>Find the integer Value of UpperName in P, till end of current section</td>
</tr>
<tr>
<td>FindNameValue</td>
<td>Search for a value from its uppercased named entry</td>
</tr>
<tr>
<td>FindNameValue</td>
<td>Search and returns a value from its uppercased named entry</td>
</tr>
<tr>
<td>FindNextUTF8WordBegin</td>
<td>Points to the beginning of the next word stored in U</td>
</tr>
<tr>
<td>FindObjectEntry</td>
<td>Retrieve a property value in a text-encoded class</td>
</tr>
<tr>
<td>FindObjectEntryWithoutExt</td>
<td>Retrieve a filename property value in a text-encoded class</td>
</tr>
<tr>
<td>FindPropName</td>
<td>Return the index of Value in Values[], using IdemPropNameU(), -1 if not found</td>
</tr>
<tr>
<td>FindPropName</td>
<td>Return the index of Value in Values[], -1 if not found</td>
</tr>
<tr>
<td>FindRawUTF8</td>
<td>Low-level efficient search of Value in Values[]</td>
</tr>
<tr>
<td>FindRawUTF8</td>
<td>Return the index of Value in Values[], -1 if not found</td>
</tr>
<tr>
<td>FindRawUTF8</td>
<td>Return the index of Value in Values[], -1 if not found</td>
</tr>
<tr>
<td>FindSectionFirstLine</td>
<td>Find the position of the [SEARCH] section in source</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FindSectionFirstLineW</td>
<td>Find the position of the [SEARCH] section in source</td>
</tr>
<tr>
<td>FindShortStringListExact</td>
<td>Fast search of an exact case-insensitive match of a RTTI's PShortString array</td>
</tr>
<tr>
<td>FindShortStringListTrimLowerCase</td>
<td>Fast case-insensitive search of a left-trimmed lowercase match of a RTTI's PShortString array</td>
</tr>
<tr>
<td>FindShortStringListTrimLowerCaseExact</td>
<td>Fast case-sensitive search of a left-trimmed lowercase match of a RTTI's PShortString array</td>
</tr>
<tr>
<td>FindUnicode</td>
<td>Return true if Upper (Unicode encoded) is contained in U^ (UTF-8 encoded)</td>
</tr>
<tr>
<td>FindUTF8</td>
<td>Return true if UpperValue (Ansi) is contained in U^ (UTF-8 encoded)</td>
</tr>
<tr>
<td>FindWinAnsiIniEntry</td>
<td>Find a Name= Value in a [Section] of a INI WinAnsi Content</td>
</tr>
<tr>
<td>FloatStrCopy</td>
<td>Copy a floating-point text buffer with proper correction and validation</td>
</tr>
<tr>
<td>FloatToJSONNan</td>
<td>Recognize if the supplied text is NAN/INF/+INF/-INF, i.e. not a number</td>
</tr>
<tr>
<td>FloatToShortNan</td>
<td>Check if the supplied text is NAN/INF/+INF/-INF, i.e. not a number</td>
</tr>
<tr>
<td>FloatToStrNan</td>
<td>Check if the supplied text is NAN/INF/+INF/-INF, i.e. not a number</td>
</tr>
<tr>
<td>fnv32</td>
<td>Simple FNV-1a hashing function</td>
</tr>
<tr>
<td>FormatBuffer</td>
<td>Fast Format() function replacement, tuned for direct memory buffer write</td>
</tr>
<tr>
<td>FormatShort</td>
<td>Fast Format() function replacement, for UTF-8 content stored in shortstring</td>
</tr>
<tr>
<td>FormatShort16</td>
<td>Fast Format() function replacement, for UTF-8 content stored in TShort16</td>
</tr>
<tr>
<td>FormatString</td>
<td>Fast Format() function replacement, tuned for small content</td>
</tr>
<tr>
<td>FormatString</td>
<td>Fast Format() function replacement, tuned for small content</td>
</tr>
<tr>
<td>FormatToShort</td>
<td>Fast Format() function replacement, for UTF-8 content stored in shortstring</td>
</tr>
<tr>
<td>FormatUTF8</td>
<td>Fast Format() function replacement, optimized for RawUTF8</td>
</tr>
<tr>
<td>FormatUTF8</td>
<td>Fast Format() function replacement, optimized for RawUTF8</td>
</tr>
<tr>
<td>FormatUTF8</td>
<td>Fast Format() function replacement, handling % and ? parameters</td>
</tr>
<tr>
<td>FormatUTF8ToVariant</td>
<td>Convert a FormatUTF8() UTF-8 encoded string into a variant RawUTF8 varString</td>
</tr>
<tr>
<td>FromI32</td>
<td>Initializes a dynamic array from a set of 32-bit integer signed values</td>
</tr>
<tr>
<td>FromI64</td>
<td>Initializes a dynamic array from a set of 64-bit integer signed values</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FromU32</td>
<td>Initializes a dynamic array from a set of 32-bit integer unsigned values</td>
</tr>
<tr>
<td>FromU64</td>
<td>Initializes a dynamic array from a set of 64-bit integer unsigned values</td>
</tr>
<tr>
<td>FromVarBlob</td>
<td>Retrieve pointer and length to a variable-length text/blob buffer</td>
</tr>
<tr>
<td>FromVarInt32</td>
<td>Convert a 32-bit variable-length integer buffer into an integer</td>
</tr>
<tr>
<td>FromVarInt64</td>
<td>Convert a 64-bit variable-length integer buffer into a Int64</td>
</tr>
<tr>
<td>FromVarInt64Value</td>
<td>Convert a 64-bit variable-length integer buffer into a Int64</td>
</tr>
<tr>
<td>FromVarString</td>
<td>Retrieve a variable-length UTF-8 encoded text buffer in a newly allocation RawUTF8</td>
</tr>
<tr>
<td>FromVarString</td>
<td>Safe retrieve a variable-length UTF-8 encoded text buffer in a newly allocation RawUTF8</td>
</tr>
<tr>
<td>FromVarString</td>
<td>Retrieve a variable-length text buffer</td>
</tr>
<tr>
<td>FromVarString</td>
<td>Retrieve a variable-length UTF-8 encoded text buffer in a temporary buffer</td>
</tr>
<tr>
<td>FromVarString</td>
<td>Retrieve a variable-length UTF-8 encoded text buffer in a temporary buffer</td>
</tr>
<tr>
<td>FromVarString</td>
<td>Retrieve a variable-length text buffer</td>
</tr>
<tr>
<td>FromVarUInt32</td>
<td>Convert a 32-bit variable-length integer buffer into a cardinal</td>
</tr>
<tr>
<td>FromVarUInt32</td>
<td>Convert a 32-bit variable-length integer buffer into a cardinal</td>
</tr>
<tr>
<td>FromVarUInt32Big</td>
<td>Convert a 32-bit variable-length integer buffer into a cardinal</td>
</tr>
<tr>
<td>FromVarUInt32High</td>
<td>Convert a 32-bit variable-length integer buffer into a cardinal</td>
</tr>
<tr>
<td>FromVarUInt32Safe</td>
<td>Safely convert a 32-bit variable-length integer buffer into a cardinal</td>
</tr>
<tr>
<td>FromVarUInt32Up128</td>
<td>Convert a 32-bit variable-length integer buffer into a cardinal</td>
</tr>
<tr>
<td>FromVarUInt64</td>
<td>Convert a 64-bit variable-length integer buffer into a UInt64</td>
</tr>
<tr>
<td>FromVarUInt64</td>
<td>Convert a 64-bit variable-length integer buffer into a UInt64</td>
</tr>
<tr>
<td>FromVarUInt64Safe</td>
<td>Safely convert a 64-bit variable-length integer buffer into a UInt64</td>
</tr>
<tr>
<td>FromVarVariant</td>
<td>Retrieve a variant value from variable-length buffer</td>
</tr>
<tr>
<td>GarbageCollectorFree</td>
<td>Force the global &quot;Garbage collector&quot; list to be released immediately</td>
</tr>
<tr>
<td>GarbageCollectorFreeAndNil</td>
<td>A global &quot;Garbage collector&quot; for some TObject global variables which must live during whole main executable process</td>
</tr>
<tr>
<td>gcd</td>
<td>Compute GCD of two integers using substraction-based Euclidean algorithm</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GetAllBits</td>
<td>Returns TRUE if all BitCount bits are set in the input 32-bit cardinal</td>
</tr>
<tr>
<td>GetBit</td>
<td>Retrieve a particular bit status from a bit array</td>
</tr>
<tr>
<td>GetBit64</td>
<td>Retrieve a particular bit status from a 64-bit integer bits (max aIndex is 63)</td>
</tr>
<tr>
<td>GetBitCSV</td>
<td>Convert a set of bit into a CSV content</td>
</tr>
<tr>
<td>GetBitPtr</td>
<td>Retrieve a particular bit status from a bit array</td>
</tr>
<tr>
<td>GetBitsCount</td>
<td>Compute the number of bits set in a bit array</td>
</tr>
<tr>
<td>GetBitsCountPas</td>
<td>Pure pascal version of GetBitsCountPtrInt()</td>
</tr>
<tr>
<td>GetBitsCountSSE42</td>
<td>SSE 4.2 version of GetBitsCountPtrInt()</td>
</tr>
<tr>
<td>GetBoolean</td>
<td>Get a boolean value stored as true/false text in P^</td>
</tr>
<tr>
<td>GetCaptionFromClass</td>
<td>UnCamelCase and translate the class name, trimming any left 'T', 'TSyn', 'TSQL' or 'TSQLRecord'</td>
</tr>
<tr>
<td>GetCaptionFromEnum</td>
<td>UnCamelCase and translate the enumeration item</td>
</tr>
<tr>
<td>GetCaptionFromPCharLen</td>
<td>UnCamelCase and translate a char buffer</td>
</tr>
<tr>
<td>GetCaptionFromTrimmed</td>
<td>Low-level helper to retrieve a (translated) caption from a PShortString</td>
</tr>
<tr>
<td>GetCardinal</td>
<td>Get the unsigned 32-bit integer value stored in P^</td>
</tr>
<tr>
<td>GetCardinalDef</td>
<td>Get the unsigned 32-bit integer value stored in P^</td>
</tr>
<tr>
<td>GetCardinalW</td>
<td>Get the unsigned 32-bit integer value stored as Unicode string in P^</td>
</tr>
<tr>
<td>GetClassParent</td>
<td>Just a wrapper around vmtParent to avoid a function call</td>
</tr>
<tr>
<td>GetCSVItem</td>
<td>Return n-th indexed CSV string in P, starting at Index=0 for first one</td>
</tr>
<tr>
<td>GetCSVItemString</td>
<td>Return n-th indexed CSV string in P, starting at Index=0 for first one</td>
</tr>
<tr>
<td>GetDelphiCompilerVersion</td>
<td>Return the Delphi/FPC Compiler Version</td>
</tr>
<tr>
<td>GetDisplayNameFromClass</td>
<td>Will get a class name as UTF-8</td>
</tr>
<tr>
<td>GetEnumCaptions</td>
<td>Helper to retrieve all (translated) caption texts of an enumerate</td>
</tr>
<tr>
<td>GetEnumName</td>
<td>Helper to retrieve the text of an enumerate item</td>
</tr>
<tr>
<td>GetEnumNames</td>
<td>Helper to retrieve all texts of an enumerate</td>
</tr>
<tr>
<td>GetEnumNameValue</td>
<td>Helper to retrieve the index of an enumerate item from its text</td>
</tr>
<tr>
<td>GetEnumNameValue</td>
<td>Helper to retrieve the index of an enumerate item from its text</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GetEnumNameValueTrimmed</td>
<td>Retrieve the index of an enumerate item from its left-trimmed text</td>
</tr>
<tr>
<td>GetEnumNameValueTrimmedExact</td>
<td>Retrieve the index of an enumerate item from its left-trimmed text</td>
</tr>
<tr>
<td>GetEnumTrimmedNames</td>
<td>Helper to retrieve all trimmed texts of an enumerate as UTF-8 strings</td>
</tr>
<tr>
<td>GetEnumTrimmedNames</td>
<td>Helper to retrieve all trimmed texts of an enumerate</td>
</tr>
<tr>
<td>GetExtended</td>
<td>Get the extended floating point value stored in P^</td>
</tr>
<tr>
<td>GetExtended</td>
<td>Get the extended floating point value stored in P^</td>
</tr>
<tr>
<td>GetFileNameExtIndex</td>
<td>Extract a file extension from a file name, then compare with a comma</td>
</tr>
<tr>
<td></td>
<td>separated list of extensions</td>
</tr>
<tr>
<td>GetFileNameWithoutExt</td>
<td>Extract file name, without its extension</td>
</tr>
<tr>
<td>GetHighUTF8UCS4</td>
<td>Internal function, used to retrieve a UCS4 char (&gt;127) from UTF-8</td>
</tr>
<tr>
<td>GetInt64</td>
<td>Get the 64-bit integer value stored in P^</td>
</tr>
<tr>
<td>GetInt64</td>
<td>Get the 64-bit signed integer value stored in P^</td>
</tr>
<tr>
<td>GetInt64Def</td>
<td>Get the 64-bit integer value stored in P^</td>
</tr>
<tr>
<td>GetInteger</td>
<td>Get the signed 32-bit integer value stored in P^</td>
</tr>
<tr>
<td>GetInteger</td>
<td>Get the signed 32-bit integer value stored in P^..PEnd^</td>
</tr>
<tr>
<td>GetIntegerDef</td>
<td>Get the signed 32-bit integer value stored in P^</td>
</tr>
<tr>
<td>GetIntegerDef</td>
<td>Get the signed 32-bit integer value stored in P^</td>
</tr>
<tr>
<td>GetJpegSize</td>
<td>Fast guess of the size, in pixels, of a JPEG memory buffer</td>
</tr>
<tr>
<td>GetJpegSize</td>
<td>Fast guess of the size, in pixels, of a JPEG file</td>
</tr>
<tr>
<td>GetJSONField</td>
<td>Efficient JSON field in-place decoding, within a UTF-8 encoded buffer</td>
</tr>
<tr>
<td>GetJSONFieldOrObjectOrArray</td>
<td>Decode a JSON content in an UTF-8 encoded buffer</td>
</tr>
<tr>
<td>GetJSONItemAsRawJSON</td>
<td>Retrieve the next JSON item as a RawJSON variable</td>
</tr>
<tr>
<td>GetJSONItemAsRawUTF8</td>
<td>Retrieve the next JSON item as a RawUTF8 decoded buffer</td>
</tr>
<tr>
<td>GetJSONPropName</td>
<td>Decode a JSON field name in an UTF-8 encoded shortstring variable</td>
</tr>
<tr>
<td>GetJSONPropName</td>
<td>Decode a JSON field name in an UTF-8 encoded buffer</td>
</tr>
<tr>
<td>GetLastCSVItem</td>
<td>Return last CSV string in the supplied UTF-8 content</td>
</tr>
<tr>
<td>GetLineContains</td>
<td>Returns TRUE if the supplied uppcased text is contained in the text buffer</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>GetLineSize</td>
<td>Compute the line length from source array of chars</td>
</tr>
<tr>
<td>GetLineSizeSmallerThan</td>
<td>Returns true if the line length from source array of chars is not less than the specified count</td>
</tr>
<tr>
<td>GetMemAligned</td>
<td>Initialize a RawByteString, ensuring returned “aligned” pointer is 16-bytes aligned</td>
</tr>
<tr>
<td>GetMimeTypeContentType</td>
<td>Retrieve the MIME content type from its file name or a supplied binary buffer</td>
</tr>
<tr>
<td>GetMimeTypeContentTypeFromBuffer</td>
<td>Retrieve the MIME content type from a supplied binary buffer</td>
</tr>
<tr>
<td>GetMimeTypeContentTypeHeader</td>
<td>Retrieve the HTTP header for MIME content type from a supplied binary buffer</td>
</tr>
<tr>
<td>GetNextFieldProp</td>
<td>Retrieve the next SQL-like identifier within the UTF-8 buffer</td>
</tr>
<tr>
<td>GetNextFieldPropSameLine</td>
<td>Retrieve the next identifier within the UTF-8 buffer on the same line</td>
</tr>
<tr>
<td>GetNextItem</td>
<td>Return next CSV string (unquoted if needed) from P</td>
</tr>
<tr>
<td>GetNextItem</td>
<td>Return next CSV string from P</td>
</tr>
<tr>
<td>GetNextItem</td>
<td>Return next CSV string from P</td>
</tr>
<tr>
<td>GetNextItemCardinal</td>
<td>Return next CSV string as unsigned integer from P, 0 if no more</td>
</tr>
<tr>
<td>GetNextItemCardinalStr</td>
<td>Return next CSV string as unsigned integer from P, 0 if no more</td>
</tr>
<tr>
<td>GetNextItemCardinalW</td>
<td>Return next CSV string as unsigned integer from P, 0 if no more</td>
</tr>
<tr>
<td>GetNextItemCurrency</td>
<td>Return next CSV string as currency from P, 0.0 if no more</td>
</tr>
<tr>
<td>GetNextItemCurrency</td>
<td>Return next CSV string as currency from P, 0.0 if no more</td>
</tr>
<tr>
<td>GetNextItemDouble</td>
<td>Return next CSV string as double from P, 0.0 if no more</td>
</tr>
<tr>
<td>GetNextItemHexa</td>
<td>Return next CSV hexadecimal string as 64-bit unsigned integer from P</td>
</tr>
<tr>
<td>GetNextItemHexDisplayToBin</td>
<td>Decode next CSV hexadecimal string from P, nil if no more or not matching BinBytes</td>
</tr>
<tr>
<td>GetNextItemInt64</td>
<td>Return next CSV string as 64-bit signed integer from P, 0 if no more</td>
</tr>
<tr>
<td>GetNextItemInteger</td>
<td>Return next CSV string as signed integer from P, 0 if no more</td>
</tr>
<tr>
<td>GetNextItemQWord</td>
<td>Return next CSV string as 64-bit unsigned integer from P, 0 if no more</td>
</tr>
<tr>
<td>GetNextItemShortString</td>
<td>Return next CSV string from P, nil if no more</td>
</tr>
<tr>
<td>GetNextItemString</td>
<td>Return next CSV string from P, nil if no more</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GetNextItemToVariant</td>
<td>Convert the next CSV item from an UTF-8 encoded text buffer into a variant number or RawUTF8 varString</td>
</tr>
<tr>
<td>GetNextItemTrimed</td>
<td>Return trimmed next CSV string from P</td>
</tr>
<tr>
<td>GetNextItemTrimedCRLF</td>
<td>Return next CRLF separated value string from P, ending #10 or #13#10 trimmed</td>
</tr>
<tr>
<td>GetNextLine</td>
<td>Extract a line from source array of chars</td>
</tr>
<tr>
<td>GetNextStringLineToRawUnicode</td>
<td>Return next string delimited with #13#10 from P, nil if no more</td>
</tr>
<tr>
<td>GetNextTChar64</td>
<td>Return next CSV string from P as a #0-ended buffer, false if no more</td>
</tr>
<tr>
<td>GetNextUTF8Upper</td>
<td>Retrieve the next UCS4 value stored in U, then update the U pointer</td>
</tr>
<tr>
<td>GetNumericVariantFromJSON</td>
<td>Low-level function to set a numerical variant from an unescaped JSON number</td>
</tr>
<tr>
<td>GetPublishedMethods</td>
<td>Retrieve published methods information about any class instance</td>
</tr>
<tr>
<td>GetQWord</td>
<td>Get the 64-bit unsigned integer value stored in P^</td>
</tr>
<tr>
<td>GetSectionContent</td>
<td>Retrieve the whole content of a section as a string</td>
</tr>
<tr>
<td>GetSectionContent</td>
<td>Retrieve the whole content of a section as a string</td>
</tr>
<tr>
<td>GetSetBaseEnum</td>
<td>Low-level helper to retrive the base enumeration RTTI of a given set</td>
</tr>
<tr>
<td>GetSetName</td>
<td>Helper to retrieve the CSV text of all enumerate items defined in a set</td>
</tr>
<tr>
<td>GetSetNameShort</td>
<td>Helper to retrieve the CSV text of all enumerate items defined in a set</td>
</tr>
<tr>
<td>GetSetNameValue</td>
<td>Helper to retrieve the bit mapped integer value of a set from its JSON text</td>
</tr>
<tr>
<td>GetSystemPath</td>
<td>Returns an operating system folder</td>
</tr>
<tr>
<td>GetUnQuoteCSVItem</td>
<td>Return n-th indexed CSV string (unquoted if needed) in P, starting at Index=0 for first one</td>
</tr>
<tr>
<td>GetUTF8Char</td>
<td>Get the WideChar stored in P^ (decode UTF-8 if necessary)</td>
</tr>
<tr>
<td>GetVariantFromJSON</td>
<td>Low-level function to set a variant from an unescaped JSON number or string</td>
</tr>
<tr>
<td>GetVariantFromNotStringJSON</td>
<td>Low-level function to set a variant from an unescaped JSON non string</td>
</tr>
<tr>
<td>GlobalLock</td>
<td>Enter a giant lock for thread-safe shared process</td>
</tr>
<tr>
<td>GlobalUnlock</td>
<td>Release the giant lock for thread-safe shared process</td>
</tr>
<tr>
<td>GotoEndJSONItem</td>
<td>Reach positon just after the current JSON item in the supplied UTF-8 buffer</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>GotoEndOfJSONString</td>
<td>Get the next character after a quoted buffer</td>
</tr>
<tr>
<td>GotoEndOfQuotedString</td>
<td>Get the next character after a quoted buffer</td>
</tr>
<tr>
<td>GotoNextJSONItem</td>
<td>Reach the position of the next JSON item in the supplied UTF-8 buffer</td>
</tr>
<tr>
<td>GotoNextJSONObjectOrArray</td>
<td>Reach the position of the next JSON object of JSON array</td>
</tr>
<tr>
<td>GotoNextJSONObjectOrArray</td>
<td>Reach the position of the next JSON object of JSON array</td>
</tr>
<tr>
<td>GotoNextJSONObjectOrArrayMax</td>
<td>Reach the position of the next JSON object of JSON array</td>
</tr>
<tr>
<td>GotoNextJSONPropName</td>
<td>Read the position of the JSON value just after a property identifier</td>
</tr>
<tr>
<td>GotoNextLine</td>
<td>Fast go to next text line, ended by #13 or #13#10</td>
</tr>
<tr>
<td>GotoNextNotSpace</td>
<td>Get the next character not in [#1..'</td>
</tr>
<tr>
<td>GotoNextNotSpaceSameLine</td>
<td>Get the next character not in [#9,'']</td>
</tr>
<tr>
<td>GotoNextSpace</td>
<td>Get the next character in [#1..']</td>
</tr>
<tr>
<td>GotoNextVarInt</td>
<td>Jump a value in the 32-bit or 64-bit variable-length integer buffer</td>
</tr>
<tr>
<td>GotoNextVarString</td>
<td>Jump a value in variable-length text buffer</td>
</tr>
<tr>
<td>GUIDToRawUTF8</td>
<td>Convert a TGUID into UTF-8 encoded text</td>
</tr>
<tr>
<td>GUIDToShort</td>
<td>Convert a TGUID into text</td>
</tr>
<tr>
<td>GUIDToShort</td>
<td>Convert a TGUID into text</td>
</tr>
<tr>
<td>GUIDToString</td>
<td>Convert a TGUID into text</td>
</tr>
<tr>
<td>GUIDToText</td>
<td>Append a TGUID binary content as text</td>
</tr>
<tr>
<td>Hash128</td>
<td>Hash one THash128 value with the supplied Hasher() function</td>
</tr>
<tr>
<td>Hash128Index</td>
<td>Fast O(n) search of a 128-bit item in an array of such values</td>
</tr>
<tr>
<td>Hash256</td>
<td>Hash one THash256 value with the supplied Hasher() function</td>
</tr>
<tr>
<td>Hash256Index</td>
<td>Fast O(n) search of a 256-bit item in an array of such values</td>
</tr>
<tr>
<td>Hash32</td>
<td>Our custom efficient 32-bit hash/checksum function</td>
</tr>
<tr>
<td>Hash32</td>
<td>Our custom efficient 32-bit hash/checksum function</td>
</tr>
<tr>
<td>Hash512</td>
<td>Hash one THash512 value with the supplied Hasher() function</td>
</tr>
<tr>
<td>Hash AnsiString</td>
<td>Hash one AnsiString content with the supplied Hasher() function</td>
</tr>
<tr>
<td>Hash AnsiStringI</td>
<td>Case-insensitive hash one AnsiString content with the supplied Hasher() function</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HashByte</td>
<td>Hash one Byte value</td>
</tr>
<tr>
<td>HashFile</td>
<td>Compute the 32-bit default hash of a file content</td>
</tr>
<tr>
<td>HashInt64</td>
<td>Hash one Int64/Qword value with the supplied Hasher() function</td>
</tr>
<tr>
<td>HashInteger</td>
<td>Hash one Integer/cardinal value - simply return the value ignore Hasher() parameter</td>
</tr>
<tr>
<td>HashPointer</td>
<td>Hash one pointer value with the supplied Hasher() function</td>
</tr>
<tr>
<td>HashPtrUInt</td>
<td>Hash one PtrUInt (=NativeUInt) value with the supplied Hasher() function</td>
</tr>
<tr>
<td>HashSynUnicode</td>
<td>Hash one SynUnicode content with the supplied Hasher() function</td>
</tr>
<tr>
<td>HashSynUnicodeI</td>
<td>Case-insensitive hash one SynUnicode content with the supplied Hasher() function</td>
</tr>
<tr>
<td>HashVariant</td>
<td>Case-sensitive hash one variant content with the supplied Hasher() function</td>
</tr>
<tr>
<td>HashVariantI</td>
<td>Case-insensitive hash one variant content with the supplied Hasher() function</td>
</tr>
<tr>
<td>HashWideString</td>
<td>Hash one WideString content with the supplied Hasher() function</td>
</tr>
<tr>
<td>HashWideStringI</td>
<td>Case-insensitive hash one WideString content with the supplied Hasher() function</td>
</tr>
<tr>
<td>HashWord</td>
<td>Hash one Word value</td>
</tr>
<tr>
<td>HexDisplayToBin</td>
<td>Fast conversion from hexa chars into a binary buffer</td>
</tr>
<tr>
<td>HexDisplayToCardinal</td>
<td>Fast conversion from hexa chars into a cardinal</td>
</tr>
<tr>
<td>HexDisplayToInt64</td>
<td>Inline gives an error under release conditions with FPC fast conversion from hexa chars into a cardinal</td>
</tr>
<tr>
<td>HexDisplayToInt64</td>
<td>Fast conversion from hexa chars into a cardinal</td>
</tr>
<tr>
<td>HexToBin</td>
<td>Fast conversion from hexa chars into binary data</td>
</tr>
<tr>
<td>HexToBin</td>
<td>Fast conversion from hexa chars into binary data</td>
</tr>
<tr>
<td>HexToBinFast</td>
<td>Fast conversion with no validity check from hexa chars into binary data</td>
</tr>
<tr>
<td>HexToChar</td>
<td>Fast conversion from one hexa char pair into a 8 bit AnsiChar</td>
</tr>
<tr>
<td>HexToCharValid</td>
<td>Fast conversion from one hexa char pair into a 8 bit AnsiChar</td>
</tr>
<tr>
<td>HexToWideChar</td>
<td>Fast conversion from two hexa bytes into a 16 bit UTF-16 WideChar</td>
</tr>
<tr>
<td>IdemFileExt</td>
<td>Returns true if the file name extension contained in p^ is the same same as extup^</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IdemFileExts</td>
<td>Returns matching file name extension index as extup^</td>
</tr>
<tr>
<td>IdemPChar</td>
<td>Returns true if the beginning of p^ is the same as up^</td>
</tr>
<tr>
<td>IdemPCharAndGetNextItem</td>
<td>Return true if IdemPChar(source,searchUp), and retrieve the value item</td>
</tr>
<tr>
<td>IdemPCharAndGetNextLine</td>
<td>Return true if IdemPChar(source,searchUp), and go to the next line of source</td>
</tr>
<tr>
<td>IdemPCharArray</td>
<td>Returns the index of a matching beginning of p^ in upArray[]</td>
</tr>
<tr>
<td>IdemPCharArray</td>
<td>Returns the index of a matching beginning of p^ in upArray two characters</td>
</tr>
<tr>
<td>IdemPCharU</td>
<td>Returns true if the beginning of p^ is the same as up^</td>
</tr>
<tr>
<td>IdemPCharW</td>
<td>Returns true if the beginning of p^ is same as up^</td>
</tr>
<tr>
<td>IdemPCharWithoutWhiteSpace</td>
<td>Returns true if the beginning of p^ is the same as up^, ignoring white spaces</td>
</tr>
<tr>
<td>IdemPropName</td>
<td>Case insensitive comparison of ASCII identifiers</td>
</tr>
<tr>
<td>IdemPropName</td>
<td>Case insensitive comparison of ASCII identifiers</td>
</tr>
<tr>
<td>IdemPropName</td>
<td>Case insensitive comparison of ASCII identifiers</td>
</tr>
<tr>
<td>IdemPropNameU</td>
<td>Case insensitive comparison of ASCII identifiers</td>
</tr>
<tr>
<td>IdemPropNameU</td>
<td>Case insensitive comparison of ASCII identifiers</td>
</tr>
<tr>
<td>IdemPropNameUSameLen</td>
<td>Case insensitive comparison of ASCII identifiers of same length</td>
</tr>
<tr>
<td>IncludeInt64</td>
<td>Ensure some 64-bit integer from Values[] will only contain Included[]</td>
</tr>
<tr>
<td>IncludeInteger</td>
<td>Ensure some 32-bit integer from Values[] will only contain Included[]</td>
</tr>
<tr>
<td>IncludeTrailingURIDelimiter</td>
<td>Ensure the supplied URI contains a trailing '/' character</td>
</tr>
<tr>
<td>InitializeCriticalSectionIfNeededAndEnter</td>
<td>On need initialization of a mutex, then enter the lock</td>
</tr>
<tr>
<td>InsertInteger</td>
<td>Insert an integer value at the specified index position of a dynamic array of integers</td>
</tr>
<tr>
<td>Int18ToChars3</td>
<td>Compute the value as encoded by TTextWriter.AddInt18ToChars3() method</td>
</tr>
<tr>
<td>Int18ToChars3</td>
<td>Compute the value as encoded by TTextWriter.AddInt18ToChars3() method</td>
</tr>
<tr>
<td>Int32ToUTF8</td>
<td>Use our fast RawUTF8 version of IntToStr()</td>
</tr>
<tr>
<td>Int32Utf8</td>
<td>Use our fast RawUTF8 version of IntToStr()</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Int64DynArrayToCSV</td>
<td>Return the corresponding CSV text from a dynamic array of 64-bit integers</td>
</tr>
<tr>
<td>Int64DynArrayToCSV</td>
<td>Return the corresponding CSV text from a dynamic array of 64-bit integers</td>
</tr>
<tr>
<td>Int64Scan</td>
<td>Fast search of an integer position in a 64-bit integer array</td>
</tr>
<tr>
<td>Int64ScanExists</td>
<td>Fast search of an integer value in a 64-bit integer array</td>
</tr>
<tr>
<td>Int64ScanIndex</td>
<td>Fast search of an integer position in a signed 64-bit integer array</td>
</tr>
<tr>
<td>Int64ToHex</td>
<td>Fast conversion from a Int64 value into hexa chars, ready to be displayed</td>
</tr>
<tr>
<td>Int64ToHex</td>
<td>Fast conversion from a Int64 value into hexa chars, ready to be displayed</td>
</tr>
<tr>
<td>Int64ToHexShort</td>
<td>Fast conversion from a Int64 value into hexa chars, ready to be displayed</td>
</tr>
<tr>
<td>IntegerDynArrayLoadFr</td>
<td>Wrap an Integer dynamic array BLOB content as stored by TDynArray.SaveTo</td>
</tr>
<tr>
<td>IntegerDynArrayToCSV</td>
<td>Return the corresponding CSV text from a dynamic array of 32-bit integer</td>
</tr>
<tr>
<td>IntegerDynArrayToCSV</td>
<td>Return the corresponding CSV text from a dynamic array of 32-bit integer</td>
</tr>
<tr>
<td>IntegerScan</td>
<td>Fast search of an unsigned integer position in an integer array</td>
</tr>
<tr>
<td>IntegerScanExists</td>
<td>Fast search of an unsigned integer position in an integer array</td>
</tr>
<tr>
<td>IntegerScanIndex</td>
<td>Fast search of an unsigned integer position in an integer array</td>
</tr>
<tr>
<td>InterfaceArrayAdd</td>
<td>Wrapper to add an item to a T*InterfaceArray dynamic array storage</td>
</tr>
<tr>
<td>InterfaceArrayAddOnce</td>
<td>Wrapper to add once an item to a T*InterfaceArray dynamic array storage</td>
</tr>
<tr>
<td>InterfaceArrayDelete</td>
<td>Wrapper to delete an item in a T*InterfaceArray dynamic array storage</td>
</tr>
<tr>
<td>InterfaceArrayDelete</td>
<td>Wrapper to delete an item in a T*InterfaceArray dynamic array storage</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>InterfaceArrayFind</td>
<td>Wrapper to search an item in a T*InterfaceArray dynamic array storage</td>
</tr>
<tr>
<td>InterlockedDecrement</td>
<td>Compatibility function, to be implemented according to the running CPU</td>
</tr>
<tr>
<td>InterlockedIncrement</td>
<td>Compatibility function, to be implemented according to the running CPU</td>
</tr>
<tr>
<td>IntervalTextToDateTime</td>
<td>Interval date/time conversion from simple text</td>
</tr>
<tr>
<td>IntervalTextToDateTimeVar</td>
<td>Interval date/time conversion from simple text</td>
</tr>
<tr>
<td>IntToString</td>
<td>Faster version than default SysUtils.IntToStr implementation</td>
</tr>
<tr>
<td>IntToString</td>
<td>Faster version than default SysUtils.IntToStr implementation</td>
</tr>
<tr>
<td>IntToString</td>
<td>Faster version than default SysUtils.IntToStr implementation</td>
</tr>
<tr>
<td>IntToThousandString</td>
<td>Convert an integer value into its textual representation with thousands marked</td>
</tr>
<tr>
<td>IP4Text</td>
<td>Convert a 32-bit integer (storing a IP4 address) into its full notation</td>
</tr>
<tr>
<td>IP6Text</td>
<td>Convert a 128-bit buffer (storing an IP6 address) into its full notation</td>
</tr>
<tr>
<td>IsAnsiCompatible</td>
<td>Return TRUE if the supplied text only contains 7-bits Ansi characters</td>
</tr>
<tr>
<td>IsAnsiCompatible</td>
<td>Return TRUE if the supplied buffer only contains 7-bits Ansi characters</td>
</tr>
<tr>
<td>IsAnsiCompatible</td>
<td>Return TRUE if the supplied buffer only contains 7-bits Ansi characters</td>
</tr>
<tr>
<td>IsAnsiCompatibleW</td>
<td>Return TRUE if the supplied UTF-16 buffer only contains 7-bits Ansi characters</td>
</tr>
<tr>
<td>IsAnsiCompatibleW</td>
<td>Return TRUE if the supplied UTF-16 buffer only contains 7-bits Ansi characters</td>
</tr>
<tr>
<td>IsBase64</td>
<td>Check if the supplied text is a valid Base64 encoded stream</td>
</tr>
<tr>
<td>IsBase64</td>
<td>Check if the supplied text is a valid Base64 encoded stream</td>
</tr>
<tr>
<td>IsCaseSensitive</td>
<td>Check if the supplied text has some case-insensitive ‘a’..'z','A'..'Z' chars</td>
</tr>
<tr>
<td>IsCaseSensitive</td>
<td>Check if the supplied text has some case-insensitive ‘a’..'z','A'..'Z' chars</td>
</tr>
<tr>
<td>IsContentCompressed</td>
<td>Retrieve if some content is compressed, from a supplied binary buffer</td>
</tr>
<tr>
<td>IsDirectoryWritable</td>
<td>Check if the directory is writable for the current user</td>
</tr>
<tr>
<td>IsEqual</td>
<td>Returns TRUE if all 64 bytes of both 512-bit buffers do match</td>
</tr>
<tr>
<td>IsEqual</td>
<td>Returns TRUE if all 48 bytes of both 384-bit buffers do match</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IsEqual</td>
<td>Returns TRUE if all 16 bytes of both 128-bit buffers do match</td>
</tr>
<tr>
<td>IsEqual</td>
<td>Returns TRUE if all 20 bytes of both 160-bit buffers do match</td>
</tr>
<tr>
<td>IsEqual</td>
<td>Returns TRUE if all 32 bytes of both 256-bit buffers do match</td>
</tr>
<tr>
<td>IsEqual</td>
<td>Returns TRUE if all bytes of both buffers do match</td>
</tr>
<tr>
<td>IsEqualGUID</td>
<td>Compare two TGUID values</td>
</tr>
<tr>
<td>IsEqualGUID</td>
<td>Compare two TGUID values</td>
</tr>
<tr>
<td>IsEqualGUIDArray</td>
<td>Returns the index of a matching TGUID in an array</td>
</tr>
<tr>
<td>IsFixedWidthCodePage</td>
<td>Check if a codepage should be handled by a TSynAnsiFixedWidth page</td>
</tr>
<tr>
<td>IsHex</td>
<td>Fast check if the supplied Hex buffer is an hexadecimal representation of a binary buffer of a given number of bytes</td>
</tr>
<tr>
<td>IsHTMLContentTypeTextual</td>
<td>Returns TRUE if the supplied HTML Headers contains 'Content-Type: text/...', 'Content-Type: application/json' or 'Content-Type: application/xml'</td>
</tr>
<tr>
<td>IsInitializedCriticalSection</td>
<td>Returns TRUE if the supplied mutex has been initialized</td>
</tr>
<tr>
<td>IsIso8601</td>
<td>Test if P(^\wedge) contains a valid ISO-8601 text encoded value</td>
</tr>
<tr>
<td>IsLeapYear</td>
<td>Our own fast version of the corresponding low-level RTL function</td>
</tr>
<tr>
<td>IsNullGUID</td>
<td>Check if a TGUID value contains only 0 bytes</td>
</tr>
<tr>
<td>Iso8601CheckAndDecode</td>
<td>Date/Time conversion from strict ISO-8601 content</td>
</tr>
<tr>
<td>Iso8601ToDatePUTF8Char</td>
<td>Date conversion from ISO-8601 (with no Time part)</td>
</tr>
<tr>
<td>Iso8601ToDateTime</td>
<td>Date/Time conversion from ISO-8601</td>
</tr>
<tr>
<td>Iso8601ToDateTimePUTF8Char</td>
<td>Date/Time conversion from ISO-8601</td>
</tr>
<tr>
<td>Iso8601ToDateTimePUTF8CharVar</td>
<td>Date/Time conversion from ISO-8601</td>
</tr>
<tr>
<td>Iso8601ToTimeLog</td>
<td>Convert a Iso8601 encoded string into a TTimeLog value</td>
</tr>
<tr>
<td>Iso8601ToTimeLogPUTF8Char</td>
<td>Convert a Iso8601 encoded string into a TTimeLog value</td>
</tr>
<tr>
<td>Iso8601ToTimePUTF8Char</td>
<td>Time conversion from ISO-8601 (with no Date part)</td>
</tr>
<tr>
<td>Iso8601ToTimePUTF8CharVar</td>
<td>Time conversion from ISO-8601 (with no Date part)</td>
</tr>
<tr>
<td>Iso8601ToTimePUTF8CharVar</td>
<td>Time conversion from ISO-8601 (with no Date part)</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IsRawUTF8DynArray</td>
<td>Check if the TypeInfo() points to an &quot;array of RawUTF8&quot;</td>
</tr>
<tr>
<td>IsRowID</td>
<td>Returns TRUE if the specified field name is either 'ID', either 'ROWID'</td>
</tr>
<tr>
<td>IsRowID</td>
<td>Returns TRUE if the specified field name is either 'ID', either 'ROWID'</td>
</tr>
<tr>
<td>IsRowIDShort</td>
<td>Returns TRUE if the specified field name is either 'ID', either 'ROWID'</td>
</tr>
<tr>
<td>IsSelect</td>
<td>Return true if the parameter is void or begin with a 'SELECT' SQL statement</td>
</tr>
<tr>
<td>IsString</td>
<td>Test if the supplied buffer is a &quot;string&quot; value or a numerical value (floating point or integer), according to the characters within</td>
</tr>
<tr>
<td>IsStringJSON</td>
<td>Test if the supplied buffer is a &quot;string&quot; value or a numerical value (floating or integer), according to the JSON encoding schema</td>
</tr>
<tr>
<td>IsUrlValid</td>
<td>Checks if the supplied UTF-8 text don't need URI encoding</td>
</tr>
<tr>
<td>IsValidJSON</td>
<td>Test if the supplied buffer is a correct JSON value</td>
</tr>
<tr>
<td>IsValidUTF8</td>
<td>Returns TRUE if the supplied buffer has valid UTF-8 encoding</td>
</tr>
<tr>
<td>IsValidUTF8</td>
<td>Returns TRUE if the supplied buffer has valid UTF-8 encoding</td>
</tr>
<tr>
<td>IsValidUTF8</td>
<td>Returns TRUE if the supplied buffer has valid UTF-8 encoding</td>
</tr>
<tr>
<td>IsValidUTF8WithoutControlChars</td>
<td>Returns TRUE if the supplied buffer has valid UTF-8 encoding with no #0..#31 control characters</td>
</tr>
<tr>
<td>IsValidUTF8WithoutControlChars</td>
<td>Returns TRUE if the supplied buffer has valid UTF-8 encoding with no #1..#31 control characters</td>
</tr>
<tr>
<td>IsVoid</td>
<td>Check all character within text are spaces or control chars</td>
</tr>
<tr>
<td>IsWinAnsi</td>
<td>Return TRUE if the supplied unicode buffer only contains WinAnsi characters</td>
</tr>
<tr>
<td>IsWinAnsi</td>
<td>Return TRUE if the supplied unicode buffer only contains WinAnsi characters</td>
</tr>
<tr>
<td>IsWinAnsiU</td>
<td>Return TRUE if the supplied UTF-8 buffer only contains WinAnsi characters</td>
</tr>
<tr>
<td>IsWinAnsiU8Bit</td>
<td>Return TRUE if the supplied UTF-8 buffer only contains WinAnsi 8 bit characters</td>
</tr>
<tr>
<td>IsZero</td>
<td>Returns TRUE if all 32 bytes of this 256-bit buffer equal zero</td>
</tr>
<tr>
<td>IsZero</td>
<td>Returns TRUE if all bytes equal zero</td>
</tr>
<tr>
<td>IsZero</td>
<td>Returns TRUE if Value is nil or all supplied Values[] equal 0</td>
</tr>
<tr>
<td>IsZero</td>
<td>Returns TRUE if Value is nil or all supplied Values[] equal 0</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>IsZero</td>
<td>Returns TRUE if all 48 bytes of this 384-bit buffer equal zero</td>
</tr>
<tr>
<td>IsZero</td>
<td>Returns TRUE if all 16 bytes of this 128-bit buffer equal zero</td>
</tr>
<tr>
<td>IsZero</td>
<td>Returns TRUE if Value is nil or all supplied Values[] equal &quot; &quot;</td>
</tr>
<tr>
<td>IsZero</td>
<td>Returns TRUE if all 20 bytes of this 160-bit buffer equal zero</td>
</tr>
<tr>
<td>IsZero</td>
<td>Returns TRUE if all 64 bytes of this 512-bit buffer equal zero</td>
</tr>
<tr>
<td>IsZeroSmall</td>
<td>Returns TRUE if all of a few bytes equal zero</td>
</tr>
<tr>
<td>JSONArrayCount</td>
<td>Compute the number of elements of a JSON array</td>
</tr>
<tr>
<td>JSONArrayCount</td>
<td>Compute the number of elements of a JSON array</td>
</tr>
<tr>
<td>JSONArrayDecode</td>
<td>Retrieve all elements of a JSON array</td>
</tr>
<tr>
<td>JSONArrayItem</td>
<td>Go to the #nth item of a JSON array</td>
</tr>
<tr>
<td>JSONBufferReformat</td>
<td>Formats and indents a JSON array or document to the specified layout</td>
</tr>
<tr>
<td>JSONBufferReformatToFile</td>
<td>Formats and indents a JSON array or document as a file</td>
</tr>
<tr>
<td>JSONBufferToXML</td>
<td>Convert a JSON array or document into a simple XML content</td>
</tr>
<tr>
<td>JSONDecode</td>
<td>Decode the supplied UTF-8 JSON content for the supplied names</td>
</tr>
<tr>
<td>JSONDecode</td>
<td>Decode the supplied UTF-8 JSON content for the one supplied name</td>
</tr>
<tr>
<td>JSONDecode</td>
<td>Decode the supplied UTF-8 JSON content for the supplied names</td>
</tr>
<tr>
<td>JSONDecode</td>
<td>Decode the supplied UTF-8 JSON content into an array of name/value pairs</td>
</tr>
<tr>
<td>JSONDecode</td>
<td>Decode the supplied UTF-8 JSON content for the supplied names</td>
</tr>
<tr>
<td>JSONEncode</td>
<td>Encode the supplied (extended) JSON content, with parameters, as an UTF-8 valid JSON object content</td>
</tr>
<tr>
<td>JSONEncode</td>
<td>Encode the supplied data as an UTF-8 valid JSON object content</td>
</tr>
<tr>
<td>JSONEncodeArrayDouble</td>
<td>Encode the supplied floating-point array data as a valid JSON array</td>
</tr>
<tr>
<td>JSONEncodeArrayInteger</td>
<td>Encode the supplied integer array data as a valid JSON array</td>
</tr>
<tr>
<td>JSONEncodeArrayOfConst</td>
<td>Encode the supplied array data as a valid JSON array content</td>
</tr>
<tr>
<td>JSONEncodeArrayOfConst</td>
<td>Encode the supplied array data as a valid JSON array content</td>
</tr>
<tr>
<td>JSONEncodeArrayUTF8</td>
<td>Encode the supplied RawUTF8 array data as an UTF-8 valid JSON array content</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>JSONEncodeNameSQLValue</td>
<td>Encode as JSON (&quot;name&quot;:value) object, from a potential SQL quoted value</td>
</tr>
<tr>
<td>JSONObjectAsJSONArray</td>
<td>Convert one JSON object into two JSON arrays of keys and values</td>
</tr>
<tr>
<td>JsonObjectByPath</td>
<td>Go to a property of a JSON object, by its full path, e.g. 'parent.child'</td>
</tr>
<tr>
<td>JsonObjectItem</td>
<td>Go to a named property of a JSON object</td>
</tr>
<tr>
<td>JsonObjectPropCount</td>
<td>Compute the number of fields in a JSON object</td>
</tr>
<tr>
<td>JsonObjectByPath</td>
<td>Return all matching properties of a JSON object</td>
</tr>
<tr>
<td>JsonPropNameValid</td>
<td>Returns TRUE if the given text buffer contains simple characters as recognized by JSON extended syntax</td>
</tr>
<tr>
<td>JSONReformat</td>
<td>Formats and indents a JSON array or document to the specified layout</td>
</tr>
<tr>
<td>JSONReformatToFile</td>
<td>Formats and indents a JSON array or document as a file</td>
</tr>
<tr>
<td>JSONRetrieveStringField</td>
<td>Retrieve a pointer to JSON string field content</td>
</tr>
<tr>
<td>JSONToVariant</td>
<td>Retrieve a variant value from a JSON UTF-8 text as per RFC 8259, RFC 7159, RFC 7158</td>
</tr>
<tr>
<td>JSONToVariantDynArray</td>
<td>Convert a JSON array into a dynamic array of variants</td>
</tr>
<tr>
<td>JSONToVariantInPlace</td>
<td>Retrieve a variant value from a JSON buffer as per RFC 8259, RFC 7159, RFC 7158</td>
</tr>
<tr>
<td>JSONToXML</td>
<td>Convert a JSON array or document into a simple XML content</td>
</tr>
<tr>
<td>KahanSum</td>
<td>Compute the sum of values, using a running compensation for lost low-order bits</td>
</tr>
<tr>
<td>KB</td>
<td>Delphi 2007 is buggy as hell convert a string size to a human readable value</td>
</tr>
<tr>
<td>KB</td>
<td>Convert a size to a human readable value</td>
</tr>
<tr>
<td>KB</td>
<td>Convert a size to a human readable value power-of-two metric value</td>
</tr>
<tr>
<td>KB</td>
<td>Delphi 2007 is buggy as hell convert a size to a human readable value</td>
</tr>
<tr>
<td>KBNoSpace</td>
<td>Delphi 2007 is buggy as hell convert a size to a human readable value</td>
</tr>
<tr>
<td>KBU</td>
<td>Convert a size to a human readable value</td>
</tr>
<tr>
<td>kr32</td>
<td>Standard Kernighan &amp; Ritchie hash from &quot;The C programming Language&quot;, 3rd edition</td>
</tr>
<tr>
<td>LogEscape</td>
<td>Fill TLLogEscape stack buffer with the (hexadecimal) chars of the input binary</td>
</tr>
<tr>
<td>LogEscapeFull</td>
<td>Returns a text buffer with the (hexadecimal) chars of the input binary</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>LogEscapeFull</td>
<td>Returns a text buffer with the (hexadecimal) chars of the input binary</td>
</tr>
<tr>
<td>LogToTextFile</td>
<td>Log a message to a local text file</td>
</tr>
<tr>
<td>LowerCase</td>
<td>Fast conversion of the supplied text into lowercase</td>
</tr>
<tr>
<td>LowerCaseCopy</td>
<td>Fast conversion of the supplied text into lowercase</td>
</tr>
<tr>
<td>LowerCaseSelf</td>
<td>Fast in-place conversion of the supplied variable text into lowercase</td>
</tr>
<tr>
<td>LowerCaseU</td>
<td>Fast conversion of the supplied text into 8 bit lowercase</td>
</tr>
<tr>
<td>LowerCaseUnicode</td>
<td>Accurate conversion of the supplied UTF-8 content into the corresponding lower-case Unicode characters</td>
</tr>
<tr>
<td>MaxInteger</td>
<td>Find the maximum 32-bit integer in Values[]</td>
</tr>
<tr>
<td>MedianQuickSelect</td>
<td>Compute the median of a serie of values, using &quot;Quickselect&quot;</td>
</tr>
<tr>
<td>MedianQuickSelectInteger</td>
<td>Compute the median of an integer serie of values, using &quot;Quickselect&quot;</td>
</tr>
<tr>
<td>MicroSecToString</td>
<td>Delphi 2007 is buggy as hell convert a micro seconds elapsed time into a human readable value</td>
</tr>
<tr>
<td>MicroSecToString</td>
<td>Convert a micro seconds elapsed time into a human readable value</td>
</tr>
<tr>
<td>MoveSmall</td>
<td>An alternative Move() function tuned for small unaligned counts</td>
</tr>
<tr>
<td>mul64x64</td>
<td>Fast computation of two 64-bit unsigned integers into a 128-bit value</td>
</tr>
<tr>
<td>MultiEventAdd</td>
<td>Low-level wrapper to add a callback to a dynamic list of events</td>
</tr>
<tr>
<td>MultiEventFind</td>
<td>Low-level wrapper to check if a callback is in a dynamic list of events</td>
</tr>
<tr>
<td>MultiEventMerge</td>
<td>Low-level wrapper to add one or several callbacks from another list of events</td>
</tr>
<tr>
<td>MultiEventRemove</td>
<td>Low-level wrapper to remove a callback from a dynamic list of events</td>
</tr>
<tr>
<td>MultiEventRemove</td>
<td>Low-level wrapper to remove a callback from a dynamic list of events</td>
</tr>
<tr>
<td>MultiPartFormDataAddField</td>
<td>Encode a field in a multipart array</td>
</tr>
<tr>
<td>MultiPartFormDataAddField</td>
<td>Encode a field in a multipart array</td>
</tr>
<tr>
<td>MultiPartFormDataDecode</td>
<td>Decode multipart/form-data POST request content</td>
</tr>
<tr>
<td>MultiPartFormDataEncode</td>
<td>Encode multipart fields and files</td>
</tr>
<tr>
<td>NeedsJsonEscape</td>
<td>Returns TRUE if the given text buffers would be escaped when written as JSON</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NeedsJsonEscape</td>
<td>Returns TRUE if the given text buffers would be escaped when written as JSON</td>
</tr>
<tr>
<td>NeedsJsonEscape</td>
<td>Returns TRUE if the given text buffers would be escaped when written as JSON</td>
</tr>
<tr>
<td>NewSynLocker</td>
<td>Allocate and initialize a TSynLocker instance</td>
</tr>
<tr>
<td>NextGrow</td>
<td>Compute the new capacity when expanding an array of items</td>
</tr>
<tr>
<td>NextNotSpaceCharIs</td>
<td>Check if the next character not in [#1..'] matches a given value</td>
</tr>
<tr>
<td>NextUTF8UCS4</td>
<td>Get the UCS4 char stored in P^ (decode UTF-8 if necessary)</td>
</tr>
<tr>
<td>NotifySortedIntegerChANGES</td>
<td>Compares two 32-bit signed sorted integer arrays, and call event handlers to notify the corresponding modifications in an O(n) time</td>
</tr>
<tr>
<td>NowToString</td>
<td>Retrieve the current Date, in the ISO 8601 layout, but expanded and ready to be displayed</td>
</tr>
<tr>
<td>NowUTC</td>
<td>Returns the current UTC system date and time</td>
</tr>
<tr>
<td>NowUTCToString</td>
<td>Retrieve the current UTC Date, in the ISO 8601 layout, but expanded and ready to be displayed</td>
</tr>
<tr>
<td>ObjArrayAdd</td>
<td>Wrapper to add an item to a T*ObjArray dynamic array storage</td>
</tr>
<tr>
<td>ObjArrayAddCount</td>
<td>Wrapper to add an item to a T*ObjArray dynamic array storage</td>
</tr>
<tr>
<td>ObjArrayAddFrom</td>
<td>Wrapper to add items to a T*ObjArray dynamic array storage</td>
</tr>
<tr>
<td>ObjArrayAddOnce</td>
<td>Wrapper to add once an item to a T*ObjArray dynamic array storage</td>
</tr>
<tr>
<td>ObjArrayAddOnceFrom</td>
<td></td>
</tr>
<tr>
<td>ObjArrayAppend</td>
<td>Wrapper to add and move items to a T*ObjArray dynamic array storage</td>
</tr>
<tr>
<td>ObjArrayClear</td>
<td>Wrapper to release all items stored in a T*ObjArray dynamic array</td>
</tr>
<tr>
<td>ObjArrayClear</td>
<td>Wrapper to release all items stored in a T*ObjArray dynamic array</td>
</tr>
<tr>
<td>ObjArrayClear</td>
<td>Wrapper to release all items stored in a T*ObjArray dynamic array</td>
</tr>
<tr>
<td>ObjArrayCount</td>
<td>Wrapper to count all not nil items in a T*ObjArray dynamic array storage</td>
</tr>
<tr>
<td>ObjArrayDelete</td>
<td>Wrapper to delete an item in a T*ObjArray dynamic array storage</td>
</tr>
<tr>
<td>ObjArrayDelete</td>
<td>Wrapper to delete an item in a T*ObjArray dynamic array storage</td>
</tr>
<tr>
<td>ObjArrayDelete</td>
<td>Wrapper to delete an item in a T*ObjArray dynamic array storage</td>
</tr>
<tr>
<td>ObjArrayFind</td>
<td>Wrapper to search an item in a T*ObjArray dynamic array storage</td>
</tr>
<tr>
<td>ObjArrayFind</td>
<td>Wrapper to search an item in a T*ObjArray dynamic array storage</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ObjArrayObjArrayClear</td>
<td>Wrapper to release all items stored in an array of T*ObjArray dynamic array</td>
</tr>
<tr>
<td>ObjArraysClear</td>
<td>Wrapper to release all items stored in several T*ObjArray dynamic arrays</td>
</tr>
<tr>
<td>ObjArraySetLength</td>
<td>Wrapper to set the length of a T*ObjArray dynamic array storage</td>
</tr>
<tr>
<td>ObjArraySort</td>
<td>Wrapper to sort the items stored in a T*ObjArray dynamic array</td>
</tr>
<tr>
<td>ObjArrayToJSON</td>
<td>Wrapper to serialize a T*ObjArray dynamic array as JSON</td>
</tr>
<tr>
<td>ObjectsToJSON</td>
<td>Will serialize set of TObject into its UTF-8 JSON representation</td>
</tr>
<tr>
<td>ObjectToJSON</td>
<td>Will serialize any TObject into its UTF-8 JSON representation</td>
</tr>
<tr>
<td>ObjectToVariant</td>
<td>Will convert any TObject into a TDocVariant document instance</td>
</tr>
<tr>
<td>ObjectToVariant</td>
<td>Will convert any TObject into a TDocVariant document instance</td>
</tr>
<tr>
<td>ObjectToVariant</td>
<td>Will convert any TObject into a TDocVariant document instance</td>
</tr>
<tr>
<td>OctToBin</td>
<td>Conversion from octal C-like escape into binary data</td>
</tr>
<tr>
<td>OctToBin</td>
<td>Conversion from octal C-like escape into binary data</td>
</tr>
<tr>
<td>OrMemory</td>
<td>Logical OR of two memory buffers</td>
</tr>
<tr>
<td>PatchCode</td>
<td>Self-modifying code - change some memory buffer in the code segment</td>
</tr>
<tr>
<td>PatchCodePtrUInt</td>
<td>Self-modifying code - change one PtrUInt in the code segment</td>
</tr>
<tr>
<td>Plural</td>
<td>Write count number and append ‘s’ (if needed) to form a plural English noun</td>
</tr>
<tr>
<td>PointerToHex</td>
<td>Fast conversion from a pointer data into hexa chars, ready to be displayed</td>
</tr>
<tr>
<td>PointerToHex</td>
<td>Fast conversion from a pointer data into hexa chars, ready to be displayed</td>
</tr>
<tr>
<td>PointerToHexShort</td>
<td>Fast conversion from a pointer data into hexa chars, ready to be displayed</td>
</tr>
<tr>
<td>PosChar</td>
<td>Fast retrieve the position of a given character</td>
</tr>
<tr>
<td>PosCharAny</td>
<td>Fast retrieve the position of any value of a given set of characters</td>
</tr>
<tr>
<td>PosEx</td>
<td>Faster RawUTF8 Equivalent of standard StrUtils.PosEx</td>
</tr>
<tr>
<td>PosExChar</td>
<td>Optimized version of PosEx() with search text as one AnsiChar</td>
</tr>
<tr>
<td>PosI</td>
<td>A non case-sensitive RawUTF8 version of Pos()</td>
</tr>
<tr>
<td>PosIU</td>
<td>A non case-sensitive RawUTF8 version of Pos()</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>PropNamesValid</td>
<td>Returns TRUE if the given text buffers contains A..Z,0..9,_. characters</td>
</tr>
<tr>
<td>PropNameValid</td>
<td>Returns TRUE if the given text buffer contains a..z,A..Z,0..9,_. characters</td>
</tr>
<tr>
<td>PtrArrayAdd</td>
<td>Wrapper to add an item to a array of pointer dynamic array storage</td>
</tr>
<tr>
<td>PtrArrayAddOnce</td>
<td>Wrapper to add once an item to a array of pointer dynamic array storage</td>
</tr>
<tr>
<td>PtrArrayDelete</td>
<td>Wrapper to delete an item from a array of pointer dynamic array storage</td>
</tr>
<tr>
<td>PtrArrayDelete</td>
<td>Wrapper to delete an item from a array of pointer dynamic array storage</td>
</tr>
<tr>
<td>PtrArrayFind</td>
<td>Wrapper to find an item to a array of pointer dynamic array storage</td>
</tr>
<tr>
<td>PtrUIntScan</td>
<td>Fast search of a pointer-sized unsigned integer in an pointer-sized integer array</td>
</tr>
<tr>
<td>PtrUIntScanExists</td>
<td>Fast search of a pointer-sized unsigned integer position in an pointer-sized integer array</td>
</tr>
<tr>
<td>PtrUIntScanIndex</td>
<td>Fast search of a pointer-sized unsigned integer position in an pointer-sized integer array</td>
</tr>
<tr>
<td>QuickSortCompare</td>
<td>Performs a QuickSort using a comparison callback</td>
</tr>
<tr>
<td>QuickSortIndexedPUTF8Char</td>
<td>Sort a dynamic array of PUTF8Char items, via an external array of indexes</td>
</tr>
<tr>
<td>QuickSortInt64</td>
<td>Sort a 64-bit signed Integer array, low values first</td>
</tr>
<tr>
<td>QuickSortInt64</td>
<td>Sort a 64-bit Integer array, low values first</td>
</tr>
<tr>
<td>QuickSortInteger</td>
<td>Sort an Integer array, low values first</td>
</tr>
<tr>
<td>QuickSortInteger</td>
<td>Sort an Integer array, low values first</td>
</tr>
<tr>
<td>QuickSortInteger</td>
<td>Sort an Integer array, low values first</td>
</tr>
<tr>
<td>QuickSortInteger</td>
<td>Sort an Integer array, low values first</td>
</tr>
<tr>
<td>QuickSortPointer</td>
<td>Sort a pointer array, low values first</td>
</tr>
<tr>
<td>QuickSortPtrInt</td>
<td>Sort a PtrInt array, low values first</td>
</tr>
<tr>
<td>QuickSortQWord</td>
<td>Sort a 64-bit unsigned Integer array, low values first</td>
</tr>
<tr>
<td>QuickSortRawUTF8</td>
<td>Sort a dynamic array of RawUTF8 items</td>
</tr>
<tr>
<td>QuickSortWord</td>
<td>Sort a 16 bit unsigned Integer array, low values first</td>
</tr>
<tr>
<td>QuotedStr</td>
<td>Format a text content with SQL-like quotes</td>
</tr>
<tr>
<td>QuotedStr</td>
<td>Format a text content with SQL-like quotes</td>
</tr>
<tr>
<td>QuotedStrJSON</td>
<td>Convert UTF-8 content into a JSON string</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>QuotedStrJSON</td>
<td>Convert UTF-8 buffer into a JSON string</td>
</tr>
<tr>
<td>QuotedStrJSON</td>
<td>Convert UTF-8 content into a JSON string</td>
</tr>
<tr>
<td>QWordScanIndex</td>
<td>Fast search of an integer position in an unsigned 64-bit integer array</td>
</tr>
<tr>
<td>Random32</td>
<td>Fast compute of some 32-bit random value</td>
</tr>
<tr>
<td>Random32</td>
<td>Fast compute of some 32-bit random value, with a maximum (excluded) upper value</td>
</tr>
<tr>
<td>Random32gsl</td>
<td>Fast compute of some 32-bit random value, using the gsl_rng_taus2 generator</td>
</tr>
<tr>
<td>Random32gsl</td>
<td>Fast compute of bounded 32-bit random value, using the gsl_rng_taus2 generator</td>
</tr>
<tr>
<td>Random32Seed</td>
<td>Seed the gsl_rng_taus2 Random32/Random32gsl generator</td>
</tr>
<tr>
<td>RandomGUID</td>
<td>Compute a random GUID value</td>
</tr>
<tr>
<td>RandomGUID</td>
<td>Compute a random GUID value</td>
</tr>
<tr>
<td>RawByteStringArrayConcat</td>
<td>Fast concatenation of several AnsiStrings</td>
</tr>
<tr>
<td>RawByteStringToBytes</td>
<td>Creates a TBytes from a RawByteString memory buffer</td>
</tr>
<tr>
<td>RawByteStringToStream</td>
<td>Create a TStream from a string content</td>
</tr>
<tr>
<td>RawByteStringToVariant</td>
<td>Convert a RawByteString content into a variant varString</td>
</tr>
<tr>
<td>RawObjectsClear</td>
<td>Low-level function calling FreeAndNil(o^) successively n times</td>
</tr>
<tr>
<td>RawUnicodeToString</td>
<td>Convert any Raw Unicode encoded buffer into a generic VCL Text</td>
</tr>
<tr>
<td>RawUnicodeToString</td>
<td>Convert any Raw Unicode encoded buffer into a generic VCL Text</td>
</tr>
<tr>
<td>RawUnicodeToString</td>
<td>Convert any Raw Unicode encoded string into a generic VCL Text</td>
</tr>
<tr>
<td>RawUnicodeToSynUnicode</td>
<td>Convert any Raw Unicode encoded String into a generic SynUnicode Text</td>
</tr>
<tr>
<td>RawUnicodeToSynUnicode</td>
<td>Convert any Raw Unicode encoded String into a generic SynUnicode Text</td>
</tr>
<tr>
<td>RawUnicodeToUtf8</td>
<td>Convert a RawUnicode string into a UTF-8 string</td>
</tr>
<tr>
<td>RawUnicodeToUtf8</td>
<td>Convert a RawUnicode PWideChar into a UTF-8 string</td>
</tr>
<tr>
<td>RawUnicodeToUtf8</td>
<td>Convert a RawUnicode UTF-16 PWideChar into a UTF-8 buffer</td>
</tr>
<tr>
<td>RawUnicodeToUtf8</td>
<td>Convert a RawUnicode PWideChar into a UTF-8 string</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>RawUnicodeToUtf8</td>
<td>Convert a RawUnicode PWideChar into a UTF-8 string</td>
</tr>
<tr>
<td>RawUnicodeToWinAnsi</td>
<td>Convert a RawUnicode PWideChar into a WinAnsi (code page 1252) string</td>
</tr>
<tr>
<td>RawUnicodeToWinAnsi</td>
<td>Convert a RawUnicode string into a WinAnsi (code page 1252) string</td>
</tr>
<tr>
<td>RawUnicodeToWinPChar</td>
<td>Direct conversion of a Unicode encoded buffer into a WinAnsi PAnsiChar buffer</td>
</tr>
<tr>
<td>RawUTF8ArrayToCSV</td>
<td>Return the corresponding CSV text from a dynamic array of UTF-8 strings</td>
</tr>
<tr>
<td>RawUTF8ArrayToQuotedCSV</td>
<td>Return the corresponding CSV quoted text from a dynamic array of UTF-8 strings</td>
</tr>
<tr>
<td>RawUTF8DynArrayClear</td>
<td>Low-level finalization of a dynamic array of RawUTF8</td>
</tr>
<tr>
<td>RawUTF8DynArrayEquals</td>
<td>True if both TRawUTF8DynArray are the same for a given number of items</td>
</tr>
<tr>
<td>RawUTF8DynArrayEquals</td>
<td>True if both TRawUTF8DynArray are the same</td>
</tr>
<tr>
<td>RawUTF8DynArrayLoadFromContains</td>
<td>Search in a RawUTF8 dynamic array BLOB content as stored by TDynArray.SaveTo</td>
</tr>
<tr>
<td>RawUTF8ToGUID</td>
<td>Convert some UTF-8 encoded text into a TGUID</td>
</tr>
<tr>
<td>RawUTF8ToVariant</td>
<td>Convert an UTF-8 encoded string into a variant RawUTF8 varString</td>
</tr>
<tr>
<td>RawUTF8ToVariant</td>
<td>Convert an UTF-8 encoded string into a variant RawUTF8 varString</td>
</tr>
<tr>
<td>RawUTF8ToVariant</td>
<td>Convert an UTF-8 encoded text buffer into a variant RawUTF8 varString</td>
</tr>
<tr>
<td>RawUTF8ToVariant</td>
<td>Convert an UTF-8 encoded text buffer into a variant RawUTF8 varString</td>
</tr>
<tr>
<td>RCU128</td>
<td>Thread-safe move of a 128-bit value using a simple Read-Copy-Update pattern</td>
</tr>
<tr>
<td>RCU32</td>
<td>Thread-safe move of a 32-bit value using a simple Read-Copy-Update pattern</td>
</tr>
<tr>
<td>RCU64</td>
<td>Thread-safe move of a 64-bit value using a simple Read-Copy-Update pattern</td>
</tr>
<tr>
<td>RCUPtr</td>
<td>Thread-safe move of a pointer value using a simple Read-Copy-Update pattern</td>
</tr>
<tr>
<td>ReadStringFromStream</td>
<td>Read an UTF-8 text from a TStream</td>
</tr>
<tr>
<td>RecordClear</td>
<td>Clear a record content</td>
</tr>
<tr>
<td>RecordCopy</td>
<td>Copy a record content from source to Dest</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RecordEquals</td>
<td>Check equality of two records by content</td>
</tr>
<tr>
<td>RecordLoad</td>
<td>Fill a record content from a memory buffer as saved by RecordSave()</td>
</tr>
<tr>
<td>RecordLoad</td>
<td>Fill a record content from a memory buffer as saved by RecordSave()</td>
</tr>
<tr>
<td>RecordLoadBase64</td>
<td>Read a record content from a Base-64 encoded content</td>
</tr>
<tr>
<td>RecordLoadJSON</td>
<td>Fill a record content from a JSON serialization as saved by TTextWriter.AddRecordJSON / RecordSaveJSON</td>
</tr>
<tr>
<td>RecordLoadJSON</td>
<td>Fill a record content from a JSON serialization as saved by TTextWriter.AddRecordJSON / RecordSaveJSON</td>
</tr>
<tr>
<td>RecordSave</td>
<td>Save a record content into a destination memory buffer</td>
</tr>
<tr>
<td>RecordSave</td>
<td>Save a record content into a RawByteString</td>
</tr>
<tr>
<td>RecordSave</td>
<td>Save a record content into a destination memory buffer</td>
</tr>
<tr>
<td>RecordSave</td>
<td>Save a record content into a destination memory buffer</td>
</tr>
<tr>
<td>RecordSaveBase64</td>
<td>Save a record content into a Base-64 encoded UTF-8 text content</td>
</tr>
<tr>
<td>RecordSaveBytes</td>
<td>Save a record content into a TBytes dynamic array</td>
</tr>
<tr>
<td>RecordSaveJSON</td>
<td>Save record into its JSON serialization as saved by TTextWriter.AddRecordJSON</td>
</tr>
<tr>
<td>RecordSaveLength</td>
<td>Compute the number of bytes needed to save a record content using the RecordSave() function</td>
</tr>
<tr>
<td>RecordTypeInfoSize</td>
<td>Retrieve the record size from its low-level RTTI</td>
</tr>
<tr>
<td>RecordZero</td>
<td>Initialize a record content</td>
</tr>
<tr>
<td>RedirectCode</td>
<td>Self-modifying code - add an asm JUMP to a redirected function</td>
</tr>
<tr>
<td>RedirectCodeRestore</td>
<td>Self-modifying code - restore a code from its RedirectCode() backup</td>
</tr>
<tr>
<td>RefCntDecFree</td>
<td>Low-level string/dynarray reference counter unprocess</td>
</tr>
<tr>
<td>ReleaseInternalWindow</td>
<td>Delete the window resources used to receive Windows Messages</td>
</tr>
<tr>
<td>RemoveCommentsFromJSON</td>
<td>Remove comments and trailing commas from a text buffer before passing it to JSON parser</td>
</tr>
<tr>
<td>RenameInCSV</td>
<td>Change a Value within a CSV string</td>
</tr>
<tr>
<td>ReplaceSection</td>
<td>Replace a whole [Section] content by a new content</td>
</tr>
<tr>
<td>ReplaceSection</td>
<td>Replace a whole [Section] content by a new content</td>
</tr>
<tr>
<td>ResourceSynLZToRawByteString</td>
<td>Creates a RawByteString memory buffer from an SynLZ-compressed embedded resource</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ResourceToRawByteString</td>
<td>Creates a RawByteString memory buffer from an embedded resource</td>
</tr>
<tr>
<td>Reverse</td>
<td>Fill already allocated Reversed[] so that Reversed[Values[i]] = i</td>
</tr>
<tr>
<td>SameTextU</td>
<td>SameText() overloaded function with proper UTF-8 decoding</td>
</tr>
<tr>
<td>SameValue</td>
<td>Compare to floating point values, with IEEE 754 double precision</td>
</tr>
<tr>
<td>SameValueFloat</td>
<td>Compare to floating point values, with IEEE 754 double precision</td>
</tr>
<tr>
<td>SaveJSON</td>
<td>Serialize most kind of content as JSON, using its RTTI</td>
</tr>
<tr>
<td>ScanUTF8</td>
<td>Read and store text into values[] according to fmt specifiers</td>
</tr>
<tr>
<td>ScanUTF8</td>
<td>Read text from P/PLen and store it into values[] according to fmt specifiers</td>
</tr>
<tr>
<td>SearchRecToDateTme</td>
<td>Get a file date and time, from a FindFirst/FindNext search</td>
</tr>
<tr>
<td>SearchRecValidFile</td>
<td>Check if a FindFirst/FindNext found instance is actually a file</td>
</tr>
<tr>
<td>SearchRecValidFolder</td>
<td>Check if a FindFirst/FindNext found instance is actually a folder</td>
</tr>
<tr>
<td>SetAppUserModelID</td>
<td>Under Windows 7 and later, will set an unique application-defined Application User Model ID (AppUserModelID) that identifies the current process to the taskbar</td>
</tr>
<tr>
<td>SetBit</td>
<td>Set a particular bit into a bit array</td>
</tr>
<tr>
<td>SetBit64</td>
<td>Set a particular bit into a 64-bit integer bits (max alindex is 63)</td>
</tr>
<tr>
<td>SetBitCSV</td>
<td>Retrieve the next CSV separated bit index</td>
</tr>
<tr>
<td>SetBitPtr</td>
<td>Set a particular bit into a bit array</td>
</tr>
<tr>
<td>SetCurrentThreadName</td>
<td>Name the current thread so that it would be easily identified in the IDE debugger</td>
</tr>
<tr>
<td>SetExecutableVersion</td>
<td>Initialize ExeVersion global variable, supplying a custom version number</td>
</tr>
<tr>
<td>SetExecutableVersion</td>
<td>Initialize ExeVersion global variable, supplying the version as text</td>
</tr>
<tr>
<td>SetInt64</td>
<td>Get the 64-bit signed integer value stored in P^</td>
</tr>
<tr>
<td>SetQWord</td>
<td>Get the 64-bit unsigned integer value stored in P^</td>
</tr>
<tr>
<td>SetThreadName</td>
<td>Name a thread so that it would be easily identified in the IDE debugger</td>
</tr>
<tr>
<td>SetThreadNameDefault</td>
<td>Could be used to override SetThreadNameInternal()</td>
</tr>
<tr>
<td>SetVariantByRef</td>
<td>Same as Dest := Source, but copying by reference</td>
</tr>
<tr>
<td>SetVariantByValue</td>
<td>Same as Dest := Source, but copying by value</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SetVariantNull</td>
<td>Same as Value := Null, but slightly faster</td>
</tr>
<tr>
<td>SetVariantUnRefSimple Value</td>
<td>Same as Dest := TVarData(Source) for simple values</td>
</tr>
<tr>
<td>ShortStringToAnsi7String</td>
<td>Direct conversion of an ANSI-7 shortstring into an AnsiString</td>
</tr>
<tr>
<td>ShortStringToAnsi7String</td>
<td>Direct conversion of an ANSI-7 shortstring into an AnsiString</td>
</tr>
<tr>
<td>ShortStringToUTF8</td>
<td>Direct conversion of a WinAnsi shortstring into a UTF-8 text</td>
</tr>
<tr>
<td>SimpleDynArrayLoadFrom</td>
<td>Wrap a simple dynamic array BLOB content as stored by TDynArray.SaveTo</td>
</tr>
<tr>
<td>SimpleRoundTo2Digits</td>
<td>Simple, no banker rounding of a Currency value to only 2 digits</td>
</tr>
<tr>
<td>SimpleRoundTo2DigitsCurr64</td>
<td>Simple, no banker rounding of a Currency value, stored as Int64, to only 2 digits</td>
</tr>
<tr>
<td>SleepHiRes</td>
<td>Similar to Windows sleep() API call, to be truly cross-platform</td>
</tr>
<tr>
<td>SortDynArray128</td>
<td>Compare two &quot;array of THash128&quot; elements</td>
</tr>
<tr>
<td>SortDynArray256</td>
<td>Compare two &quot;array of THash256&quot; elements</td>
</tr>
<tr>
<td>SortDynArray512</td>
<td>Compare two &quot;array of THash512&quot; elements</td>
</tr>
<tr>
<td>SortDynArrayAnsiString</td>
<td>Compare two &quot;array of AnsiString&quot; elements, with case sensitivity</td>
</tr>
<tr>
<td>SortDynArrayAnsiStringI</td>
<td>Compare two &quot;array of AnsiString&quot; elements, with no case sensitivity</td>
</tr>
<tr>
<td>SortDynArrayBoolean</td>
<td>Compare two &quot;array of boolean&quot; elements</td>
</tr>
<tr>
<td>SortDynArrayByte</td>
<td>Compare two &quot;array of byte&quot; elements</td>
</tr>
<tr>
<td>SortDynArrayCardinal</td>
<td>Compare two &quot;array of cardinal&quot; elements</td>
</tr>
<tr>
<td>SortDynArrayDouble</td>
<td>Compare two &quot;array of double&quot; elements</td>
</tr>
<tr>
<td>SortDynArrayFileName</td>
<td>Compare two &quot;array of TFileName&quot; elements, as file names</td>
</tr>
<tr>
<td>SortDynArrayInt64</td>
<td>Compare two &quot;array of Int64&quot; or &quot;array of Currency&quot; elements</td>
</tr>
<tr>
<td>SortDynArrayInteger</td>
<td>Compare two &quot;array of integer&quot; elements</td>
</tr>
<tr>
<td>SortDynArrayPointer</td>
<td>Compare two &quot;array of TObject/pointer&quot; elements</td>
</tr>
<tr>
<td>SortDynArrayPUTF8Char</td>
<td>Compare two &quot;array of PUTF8Char/PAnsiChar&quot; elements, with case sensitivity</td>
</tr>
<tr>
<td>SortDynArrayPUTF8CharI</td>
<td>Compare two &quot;array of PUTF8Char/PAnsiChar&quot; elements, with no case sensitivity</td>
</tr>
<tr>
<td>SortDynArrayQWord</td>
<td>Compare two &quot;array of QWord&quot; elements</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SortDynArrayRawByteString</td>
<td>Compare two &quot;array of RawByteString&quot; elements, with case sensitivity</td>
</tr>
<tr>
<td>SortDynArrayShortInt</td>
<td>Compare two &quot;array of shortint&quot; elements</td>
</tr>
<tr>
<td>SortDynArraySingle</td>
<td>Compare two &quot;array of single&quot; elements</td>
</tr>
<tr>
<td>SortDynArraySmallint</td>
<td>Compare two &quot;array of smallint&quot; elements</td>
</tr>
<tr>
<td>SortDynArrayString</td>
<td>Compare two &quot;array of generic string&quot; elements, with case sensitivity</td>
</tr>
<tr>
<td>SortDynArrayStringI</td>
<td>Compare two &quot;array of generic string&quot; elements, with no case sensitivity</td>
</tr>
<tr>
<td>SortDynArrayUnicodeString</td>
<td>Compare two &quot;array of WideString/UnicodeString&quot; elements, with case sensitivity</td>
</tr>
<tr>
<td>SortDynArrayUnicodeStringI</td>
<td>Compare two &quot;array of WideString/UnicodeString&quot; elements, with no case sensitivity</td>
</tr>
<tr>
<td>SortDynArrayVariant</td>
<td>Compare two &quot;array of variant&quot; elements, with case sensitivity</td>
</tr>
<tr>
<td>SortDynArrayVariantComp</td>
<td>Compare two &quot;array of variant&quot; elements, with or without case sensitivity</td>
</tr>
<tr>
<td>SortDynArrayVariantI</td>
<td>Compare two &quot;array of variant&quot; elements, with no case sensitivity</td>
</tr>
<tr>
<td>SortDynArrayWord</td>
<td>Compare two &quot;array of word&quot; elements</td>
</tr>
<tr>
<td>Split</td>
<td>Split a RawUTF8 string into two strings, according to SepStr separator</td>
</tr>
<tr>
<td>Split</td>
<td>Returns the left part of a RawUTF8 string, according to SepStr separator</td>
</tr>
<tr>
<td>Split</td>
<td>Split a RawUTF8 string into several strings, according to SepStr separator</td>
</tr>
<tr>
<td>Split</td>
<td>Split a RawUTF8 string into two strings, according to SepStr separator</td>
</tr>
<tr>
<td>SplitRight</td>
<td>Returns the last occurrence of the given SepChar separated context</td>
</tr>
<tr>
<td>SplitRights</td>
<td>Returns the last occurrence of the given SepChar separated context</td>
</tr>
<tr>
<td>SQLAddWhereAnd</td>
<td>Add a condition to a SQL WHERE clause, with an ' and ' if where is not void</td>
</tr>
<tr>
<td>SQLBegin</td>
<td>Go to the beginning of the SQL statement, ignoring all blanks and comments</td>
</tr>
<tr>
<td>StrCompFast</td>
<td>Buffer-safe version of StrComp(), to be used with PUTF8Char/PAnsiChar</td>
</tr>
<tr>
<td>StrCompIL</td>
<td>Use our fast version of StrCompIL(), to be used with PUTF8Char</td>
</tr>
<tr>
<td>StrCompL</td>
<td>Use our fast version of StrCompL(), to be used with PUTF8Char</td>
</tr>
<tr>
<td>StrCompW</td>
<td>Use our fast version of StrComp(), to be used with PWideChar</td>
</tr>
<tr>
<td>strcspnpas</td>
<td>Pure pascal version of strcspn(), to be used with PUTF8Char/PAnsiChar</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>strcspnss4e2</td>
<td>SSE 4.2 version of strcspn(), to be used with PUTF8Char/PAnsiChar</td>
</tr>
<tr>
<td>StrCurr64</td>
<td>Internal fast INTEGER Curr64 (value*10000) value to text conversion</td>
</tr>
<tr>
<td>StreamSynLZ</td>
<td>Compress a data content using the SynLZ algorithm from one stream into a file</td>
</tr>
<tr>
<td>StreamSynLZ</td>
<td>Compress a data content using the SynLZ algorithm from one stream into another</td>
</tr>
<tr>
<td>StreamSynLZComputeLen</td>
<td>Compute the real length of a given StreamSynLZ-compressed buffer</td>
</tr>
<tr>
<td>StreamToRawByteString</td>
<td>Read a TStream content into a String</td>
</tr>
<tr>
<td>StreamUnSynLZ</td>
<td>Uncompress using the SynLZ algorithm from one stream into another</td>
</tr>
<tr>
<td>StreamUnSynLZ</td>
<td>Uncompress using the SynLZ algorithm from one file into another</td>
</tr>
<tr>
<td>StrIComp</td>
<td>Use our fast version of StrIComp(), to be used with PUTF8Char/PAnsiChar</td>
</tr>
<tr>
<td>StringBufferToUtf8</td>
<td>Convert any generic VCL Text buffer into an UTF-8 encoded buffer</td>
</tr>
<tr>
<td>StringBufferToUtf8</td>
<td>Convert any generic VCL 0-terminated Text buffer into an UTF-8 string</td>
</tr>
<tr>
<td>StringDynArrayToRawUTF8DynArray</td>
<td>Convert the string dynamic array into a dynamic array of UTF-8 strings</td>
</tr>
<tr>
<td>StringFromFile</td>
<td>Read a File content into a String</td>
</tr>
<tr>
<td>StringListToRawUTF8DynArray</td>
<td>Convert the string list into a dynamic array of UTF-8 strings</td>
</tr>
<tr>
<td>StringReplaceAll</td>
<td>Fast version of StringReplace(S, OldPattern, NewPattern,[rfReplaceAll]);</td>
</tr>
<tr>
<td>StringReplaceAll</td>
<td>Fast version of several cascaded StringReplaceAll()</td>
</tr>
<tr>
<td>StringReplaceAllProcess</td>
<td>Actual replacement function called by StringReplaceAll() on first match</td>
</tr>
<tr>
<td>StringReplaceChars</td>
<td>Fast replace of a specified char by a given string</td>
</tr>
<tr>
<td>StringReplaceTabs</td>
<td>Fast replace of all #9 chars by a given string</td>
</tr>
<tr>
<td>StringToAnsi7</td>
<td>Convert any generic VCL Text into Ansi 7 bit encoded String</td>
</tr>
<tr>
<td>StringToGUID</td>
<td>Convert some generic Text into a TGUID</td>
</tr>
<tr>
<td>StringToRawUnicode</td>
<td>Convert any generic VCL Text into a Raw Unicode encoded String</td>
</tr>
<tr>
<td>StringToRawUnicode</td>
<td>Convert any generic VCL Text into a Raw Unicode encoded String</td>
</tr>
<tr>
<td>StringToSynUnicode</td>
<td>Convert any generic VCL Text into a SynUnicode encoded String</td>
</tr>
<tr>
<td>StringToSynUnicode</td>
<td>Convert any generic VCL Text into a SynUnicode encoded String</td>
</tr>
<tr>
<td>StringToUTF8</td>
<td>Convert any generic VCL Text into an UTF-8 encoded String</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>StringToUTF8</td>
<td>Convert any generic VCL Text into an UTF-8 encoded String</td>
</tr>
<tr>
<td>StringToUTF8</td>
<td>Convert any generic VCL Text buffer into an UTF-8 encoded String</td>
</tr>
<tr>
<td>StringToWinAnsi</td>
<td>Convert any generic VCL Text into WinAnsi (Win-1252) 8 bit encoded String</td>
</tr>
<tr>
<td>StrInt32</td>
<td>Internal fast integer val to text conversion</td>
</tr>
<tr>
<td>StrInt64</td>
<td>Internal fast Int64 val to text conversion</td>
</tr>
<tr>
<td>StrLenPas</td>
<td>Slower version of StrLen(), but which will never read beyond the string</td>
</tr>
<tr>
<td>StrLenW</td>
<td>Our fast version of StrLen(), to be used with PWideChar</td>
</tr>
<tr>
<td>StrPosI</td>
<td>A non case-sensitive version of Pos()</td>
</tr>
<tr>
<td>strspnpas</td>
<td>Pure pascal version of strspn(), to be used with PUTF8Char/PAnsiChar</td>
</tr>
<tr>
<td>strspnsse42</td>
<td>SSE 4.2 version of strspn(), to be used with PUTF8Char/PAnsiChar</td>
</tr>
<tr>
<td>StrToCurr64</td>
<td>Convert a string into its INTEGER Curr64 (value*10000) representation</td>
</tr>
<tr>
<td>StrToCurrency</td>
<td>Convert a string into its currency representation</td>
</tr>
<tr>
<td>StrUInt32</td>
<td>Internal fast unsigned integer val to text conversion</td>
</tr>
<tr>
<td>StrUInt64</td>
<td>Internal fast unsigned Int64 val to text conversion</td>
</tr>
<tr>
<td>SumInteger</td>
<td>Sum all 32-bit integers in Values[]</td>
</tr>
<tr>
<td>SymmetricEncrypt</td>
<td>Naive symmetric encryption scheme using a 32-bit key</td>
</tr>
<tr>
<td>SynchFolders</td>
<td>Ensure all files in Dest folder(s) do match the one in Reference</td>
</tr>
<tr>
<td>SynLZCompress</td>
<td>Deprecated function - please call AlgoSynLZ.Compress() method</td>
</tr>
<tr>
<td>SynLZCompress</td>
<td>Deprecated function - please call AlgoSynLZ.Compress() method</td>
</tr>
<tr>
<td>SynLZCompress</td>
<td>Deprecated function - please call AlgoSynLZ.Compress() method</td>
</tr>
<tr>
<td>SynLZCompressToBytes</td>
<td>Deprecated function - please call AlgoSynLZ.DecompressToBytes() method</td>
</tr>
<tr>
<td>SynLZCompressToBytes</td>
<td>Deprecated function - please call AlgoSynLZ.DecompressToBytes() method</td>
</tr>
<tr>
<td>SynLZDecompress</td>
<td>Deprecated function - please call AlgoSynLZ.Decompress() method</td>
</tr>
<tr>
<td>SynLZDecompress</td>
<td>Deprecated function - please call AlgoSynLZ.Decompress() method</td>
</tr>
<tr>
<td>SynLZDecompress</td>
<td>Deprecated function - please call AlgoSynLZ.Decompress() method</td>
</tr>
<tr>
<td>SynLZDecompress</td>
<td>Deprecated function - please call AlgoSynLZ.Decompress() method</td>
</tr>
<tr>
<td>SynLZDecompress</td>
<td>Deprecated function - please call AlgoSynLZ.Decompress() method</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>SynLZDecompressBody</td>
<td>Deprecated function - please call AlgoSynLZ.DecompressBody() method</td>
</tr>
<tr>
<td>SynLZDecompressHeader</td>
<td>Deprecated function - please call AlgoSynLZ.DecompressHeader() method</td>
</tr>
<tr>
<td>SynLZDecompressPartial</td>
<td>Deprecated function - please call AlgoSynLZ.DecompressPartial() method</td>
</tr>
<tr>
<td>SynRegisterCustomVariantType</td>
<td>Register a custom variant type to handle properties</td>
</tr>
<tr>
<td>SynUnicodeToString</td>
<td>Convert any SynUnicode encoded string into a generic VCL Text</td>
</tr>
<tr>
<td>SynUnicodeToUtf8</td>
<td>Convert a SynUnicode string into a UTF-8 string</td>
</tr>
<tr>
<td>TemporaryFileName</td>
<td>Compute an unique temporary file name</td>
</tr>
<tr>
<td>TextToGUID</td>
<td>Convert some text into its TGUID binary value</td>
</tr>
<tr>
<td>TextToVariant</td>
<td>Convert an UTF-8 encoded text buffer into a variant number or RawUTF8 varString</td>
</tr>
<tr>
<td>TextToVariantNumberType</td>
<td>Identify either varInt64, varDouble, varCurrency types following JSON format</td>
</tr>
<tr>
<td>TextToVariantNumberTypeNoDouble</td>
<td>Identify either varInt64 or varCurrency types following JSON format</td>
</tr>
<tr>
<td>TimeLogFromDateTime</td>
<td>Get TTimeLog value from a given Delphi date and time</td>
</tr>
<tr>
<td>TimeLogFromFile</td>
<td>Get TTimeLog value from a file date and time</td>
</tr>
<tr>
<td>TimeLogFromUnixTime</td>
<td>Get TTimeLog value from a given Unix seconds since epoch timestamp</td>
</tr>
<tr>
<td>TimeLogNow</td>
<td>Get TTimeLog value from current local system date and time</td>
</tr>
<tr>
<td>TimeLogNowUTC</td>
<td>Get TTimeLog value from current UTC system Date and Time</td>
</tr>
<tr>
<td>TimeLogToDateTime</td>
<td>Date/Time conversion from a TTimeLog value</td>
</tr>
<tr>
<td>TimeLogToUnixTime</td>
<td>Unix seconds since epoch timestamp conversion from a TTimeLog value</td>
</tr>
<tr>
<td>TimeToIso8601</td>
<td>Basic Time conversion into ISO-8601</td>
</tr>
<tr>
<td>TimeToIso8601PChar</td>
<td>Write a Time to P^ Ansi buffer</td>
</tr>
<tr>
<td>TimeToIso8601PChar</td>
<td>Write a Time to P^ Ansi buffer</td>
</tr>
<tr>
<td>TimeToString</td>
<td>Retrieve the current Time (without Date), in the ISO 8601 layout</td>
</tr>
<tr>
<td>TInt64DynArrayFrom</td>
<td>Quick helper to initialize a dynamic array of 64-bit integers from 32-bit values</td>
</tr>
<tr>
<td>TIntegerDynArrayFrom</td>
<td>Quick helper to initialize a dynamic array of integer from some constants</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>TIntegerDynArrayFrom64</td>
<td>Quick helper to initialize a dynamic array of integer from 64-bit integers</td>
</tr>
<tr>
<td>ToCardinal</td>
<td>Get the unsigned 32-bit cardinal value stored in a RawUTF8 string</td>
</tr>
<tr>
<td>ToDouble</td>
<td>Get a 64-bit floating-point value stored in a RawUTF8 string</td>
</tr>
<tr>
<td>ToInt64</td>
<td>Get the signed 64-bit integer value stored in a RawUTF8 string</td>
</tr>
<tr>
<td>ToInteger</td>
<td>Get the signed 32-bit integer value stored in a RawUTF8 string</td>
</tr>
<tr>
<td>ToText</td>
<td>Just a wrapper around vmtClassName to avoid a string/RawUTF8 conversion</td>
</tr>
<tr>
<td>ToText</td>
<td>Convert Intel CPU features as plain CSV text</td>
</tr>
<tr>
<td>ToText</td>
<td>Just a wrapper around vmtClassName to avoid a string/RawUTF8 conversion</td>
</tr>
<tr>
<td>ToText</td>
<td>Retrieve the text representation of a TDocVairnatKind</td>
</tr>
<tr>
<td>ToUTF8</td>
<td>Use our fast RawUTF8 version of IntToStr()</td>
</tr>
<tr>
<td>ToUTF8</td>
<td>Convert a TGUID into UTF-8 encoded text</td>
</tr>
<tr>
<td>ToUTF8</td>
<td>Convert any UTF-8 encoded shortstring Text into an UTF-8 encoded String</td>
</tr>
<tr>
<td>ToUTF8</td>
<td>Use our fast RawUTF8 version of IntToStr()</td>
</tr>
<tr>
<td>ToUTF8</td>
<td>Convert any Variant into UTF-8 encoded String</td>
</tr>
<tr>
<td>ToUTF8</td>
<td>Convert any generic VCL Text into an UTF-8 encoded String</td>
</tr>
<tr>
<td>ToVarInt32</td>
<td>Convert an integer into a 32-bit variable-length integer buffer</td>
</tr>
<tr>
<td>ToVarInt64</td>
<td>Convert a Int64 into a 64-bit variable-length integer buffer</td>
</tr>
<tr>
<td>ToVarString</td>
<td>Convert a RawUTF8 into an UTF-8 encoded variable-length buffer</td>
</tr>
<tr>
<td>ToVarUInt32</td>
<td>Convert a cardinal into a 32-bit variable-length integer buffer</td>
</tr>
<tr>
<td>ToVarUInt32Length</td>
<td>Return the number of bytes necessary to store a 32-bit variable-length integer</td>
</tr>
<tr>
<td>ToVarUInt32LengthWithData</td>
<td>Return the number of bytes necessary to store some data with a its 32-bit variable-length integer length</td>
</tr>
<tr>
<td>ToVarUInt64</td>
<td>Convert a UInt64 into a 64-bit variable-length integer buffer</td>
</tr>
<tr>
<td>TQWordDynArrayFrom</td>
<td>Quick helper to initialize a dynamic array of 64-bit integers from 32-bit values</td>
</tr>
<tr>
<td>TRawUTF8DynArrayFrom</td>
<td>Quick helper to initialize a dynamic array of RawUTF8 from some constants</td>
</tr>
<tr>
<td>Trim</td>
<td>Fast dedicated RawUTF8 version of Trim()</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TrimControlChars</td>
<td>Returns the supplied text content, without any control char</td>
</tr>
<tr>
<td>TrimCopy</td>
<td>Single-allocation (therefore faster) alternative to Trim(copy())</td>
</tr>
<tr>
<td>TrimLeft</td>
<td>Trims leading whitespace characters from the string by removing newline, space, and tab characters</td>
</tr>
<tr>
<td>TrimLeftLowerCase</td>
<td>Trim first lowercase chars ('otDone' will return 'Done' e.g.)</td>
</tr>
<tr>
<td>TrimLeftLowerCaseShor t</td>
<td>Trim first lowercase chars ('otDone' will return 'Done' e.g.)</td>
</tr>
<tr>
<td>TrimRight</td>
<td>Trims trailing whitespace characters from the string by removing trailing newline, space, and tab characters</td>
</tr>
<tr>
<td>TruncTo2Digits</td>
<td>Truncate a Currency value to only 2 digits</td>
</tr>
<tr>
<td>TruncTo2Digits64</td>
<td>Truncate a Currency value, stored as Int64, to only 2 digits</td>
</tr>
<tr>
<td>TruncTo2DigitsCurr64</td>
<td>Truncate a Currency value, stored as Int64, to only 2 digits</td>
</tr>
<tr>
<td>TryEncodeDate</td>
<td>Our own fast version of the corresponding low-level RTL function</td>
</tr>
<tr>
<td>TypeInfoToHash</td>
<td>Compute a crc32c-based hash of the RTTI for a managed given type</td>
</tr>
<tr>
<td>TypeInfoToName</td>
<td>Retrieve the type name from its low-level RTTI</td>
</tr>
<tr>
<td>TypeInfoToQualifiedNa me</td>
<td>Retrieve the un unit name and type name from its low-level RTTI</td>
</tr>
<tr>
<td>TypeInfoToRttiType</td>
<td>Recognize a simple type from a supplied type information</td>
</tr>
<tr>
<td>UCS4ToUTF8</td>
<td>UTF-8 encode one UCS4 character into Dest</td>
</tr>
<tr>
<td>UInt2DigitsToShort</td>
<td>Creates a 2 digits short string from a 0..99 value</td>
</tr>
<tr>
<td>UInt2DigitsToShortFast</td>
<td>Creates a 2 digits short string from a 0..99 value</td>
</tr>
<tr>
<td>UInt32ToUtf8</td>
<td>Optimized conversion of a cardinal into RawUTF8</td>
</tr>
<tr>
<td>UInt32ToUtf8</td>
<td>Optimized conversion of a cardinal into RawUTF8</td>
</tr>
<tr>
<td>UInt3DigitsToShort</td>
<td>Creates a 3 digits short string from a 0..999 value</td>
</tr>
<tr>
<td>UInt3DigitsToUTF8</td>
<td>Creates a 3 digits string from a 0..999 value as '000'..'999'</td>
</tr>
<tr>
<td>UInt4DigitsToShort</td>
<td>Creates a 4 digits short string from a 0..9999 value</td>
</tr>
<tr>
<td>UInt4DigitsToUTF8</td>
<td>Creates a 4 digits string from a 0..9999 value as '0000'..'9999'</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UInt64ToUtf8</td>
<td>Fast RawUTF8 version of IntToStr(), with proper QWord conversion</td>
</tr>
<tr>
<td>UnCamelCase</td>
<td>Convert a CamelCase string into a space separated one</td>
</tr>
<tr>
<td>UnicodeBufferToString</td>
<td>Convert an Unicode buffer into a generic VCL string</td>
</tr>
<tr>
<td>UnicodeBufferToWinAnsi</td>
<td>Convert an Unicode buffer into a WinAnsi (code page 1252) string</td>
</tr>
<tr>
<td>UniqueRawUTF8</td>
<td>Equivalence to @UTF8[1] expression to ensure a RawUTF8 variable is unique</td>
</tr>
<tr>
<td>UniqueRawUTF8ZeroToTilde</td>
<td>Will fast replace all #0 chars as ~</td>
</tr>
<tr>
<td>UnixMSTimePeriodToString</td>
<td>Delphi 2007 is buggy as hell convert some millisecond-based c-encoded time to the ISO 8601 text layout, as time or date elapsed period</td>
</tr>
<tr>
<td>UnixMSTimeToDateTime</td>
<td>Convert a millisecond-based c-encoded time (from Unix epoch 1/1/1970) as TDateTime</td>
</tr>
<tr>
<td>UnixMSTimeToFileShort</td>
<td>Convert some millisecond-based c-encoded time (from Unix epoch 1/1/1970) to a small text layout, trimming to the second resolution, perfect e.g. for naming a local file</td>
</tr>
<tr>
<td>UnixMSTimeToString</td>
<td>Convert some millisecond-based c-encoded time (from Unix epoch 1/1/1970) to the ISO 8601 text layout, including milliseconds</td>
</tr>
<tr>
<td>UnixMSTimeUTC</td>
<td>Returns the current UTC date/time as a millisecond-based c-encoded time</td>
</tr>
<tr>
<td>UnixMSTimeUTCFast</td>
<td>Returns the current UTC date/time as a millisecond-based c-encoded time</td>
</tr>
<tr>
<td>UnixTimePeriodToString</td>
<td>Delphi 2007 is buggy as hell convert some second-based c-encoded time to the ISO 8601 text layout, either as time or date elapsed period</td>
</tr>
<tr>
<td>UnixTimeToDateTime</td>
<td>Convert a second-based c-encoded time as TDateTime</td>
</tr>
<tr>
<td>UnixTimeToFileShort</td>
<td>Convert some second-based c-encoded time (from Unix epoch 1/1/1970) to a small text layout, perfect e.g. for naming a local file</td>
</tr>
<tr>
<td>UnixTimeToFileShort</td>
<td>Convert some second-based c-encoded time (from Unix epoch 1/1/1970) to a small text layout, perfect e.g. for naming a local file</td>
</tr>
<tr>
<td>UnixTimeToString</td>
<td>Convert some second-based c-encoded time (from Unix epoch 1/1/1970) to the ISO 8601 text layout</td>
</tr>
<tr>
<td>UnixTimeUTC</td>
<td>Returns the current UTC date/time as a second-based c-encoded time</td>
</tr>
<tr>
<td>UnQuotedSQLSymbolName</td>
<td>Unquote a SQL-compatible symbol name</td>
</tr>
<tr>
<td>UnQuoteSQLString</td>
<td>Unquote a SQL-compatible string</td>
</tr>
<tr>
<td>UnQuoteSQLStringVar</td>
<td>Unquote a SQL-compatible string</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UnSetBit</td>
<td>Unset/clear a particular bit into a bit array</td>
</tr>
<tr>
<td>UnSetBit64</td>
<td>Unset/clear a particular bit into a 64-bit integer bits (max alindex is 63)</td>
</tr>
<tr>
<td>UnSetBitPtr</td>
<td>Unset/clear a particular bit into a bit array</td>
</tr>
<tr>
<td>UpdateIniEntry</td>
<td>Update a Name= Value in a [Section] of a INI RawUTF8 Content</td>
</tr>
<tr>
<td>UpdateIniEntryFile</td>
<td>Update a Name= Value in a [Section] of a .INI file</td>
</tr>
<tr>
<td>UpdateIniNameValue</td>
<td>Replace a value from a given set of name=value lines</td>
</tr>
<tr>
<td>UpperCase</td>
<td>Fast conversion of the supplied text into uppercase</td>
</tr>
<tr>
<td>UpperCaseCopy</td>
<td>Fast conversion of the supplied text into uppercase</td>
</tr>
<tr>
<td>UpperCaseCopy</td>
<td>Fast conversion of the supplied text into uppercase</td>
</tr>
<tr>
<td>UpperCaseSelf</td>
<td>Fast in-place conversion of the supplied variable text into uppercase</td>
</tr>
<tr>
<td>UpperCaseU</td>
<td>Fast conversion of the supplied text into 8 bit uppercase</td>
</tr>
<tr>
<td>UpperCaseUnicode</td>
<td>Accurate conversion of the supplied UTF-8 content into the corresponding upper-case Unicode characters</td>
</tr>
<tr>
<td>UpperCopy</td>
<td>Copy source into dest(^^) with 7 bits upper case conversion</td>
</tr>
<tr>
<td>UpperCopy255</td>
<td>Copy source into a 256 chars dest(^^) buffer with 7 bits upper case conversion</td>
</tr>
<tr>
<td>UpperCopy255BufPas</td>
<td>Copy source(^^) into a 256 chars dest(^^) buffer with 7 bits upper case conversion</td>
</tr>
<tr>
<td>UpperCopy255BufSSE42</td>
<td>SSE 4.2 version of UpperCopy255Buf()</td>
</tr>
<tr>
<td>UpperCopy255W</td>
<td>Copy WideChar source into dest(^^) with upper case conversion</td>
</tr>
<tr>
<td>UpperCopy255W</td>
<td>Copy WideChar source into dest(^^) with upper case conversion</td>
</tr>
<tr>
<td>UpperCopyShort</td>
<td>Copy source into dest(^^) with 7 bits upper case conversion</td>
</tr>
<tr>
<td>UpperCopyWin255</td>
<td>Copy source into dest(^^) with WinAnsi 8 bits upper case conversion</td>
</tr>
<tr>
<td>UriDecode</td>
<td>Decode a string compatible with URI encoding into its original value</td>
</tr>
<tr>
<td>UriDecode</td>
<td>Decode a string compatible with URI encoding into its original value</td>
</tr>
<tr>
<td>UriDecodeCardinal</td>
<td>Decode a specified parameter compatible with URI encoding into its original cardinal numerical value</td>
</tr>
<tr>
<td>UriDecodeDouble</td>
<td>Decode a specified parameter compatible with URI encoding into its original floating-point value</td>
</tr>
<tr>
<td>UriDecodeExtended</td>
<td>Decode a specified parameter compatible with URI encoding into its original floating-point value</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UrlDecodeInt64</td>
<td>Decode a specified parameter compatible with URI encoding into its original Int64 numerical value</td>
</tr>
<tr>
<td>UrlDecodeInteger</td>
<td>Decode a specified parameter compatible with URI encoding into its original integer numerical value</td>
</tr>
<tr>
<td>UrlDecodeNeedParameters</td>
<td>Returns TRUE if all supplied parameters do exist in the URI encoded text</td>
</tr>
<tr>
<td>UrlDecodeNextName</td>
<td>Decode a URI-encoded Name from an input buffer</td>
</tr>
<tr>
<td>UrlDecodeNextNameValue</td>
<td>Decode the next Name=Value&amp;.... pair from input URI</td>
</tr>
<tr>
<td>UrlDecodeNextValue</td>
<td>Decode a URI-encoded Value from an input buffer</td>
</tr>
<tr>
<td>UrlDecodeValue</td>
<td>Decode a specified parameter compatible with URI encoding into its original textual value</td>
</tr>
<tr>
<td>UrlEncode</td>
<td>Encode a string to be compatible with URI encoding</td>
</tr>
<tr>
<td>UrlEncode</td>
<td>Encode supplied parameters to be compatible with URI encoding</td>
</tr>
<tr>
<td>UrlEncode</td>
<td>Encode a string to be compatible with URI encoding</td>
</tr>
<tr>
<td>UrlEncodeJsonObject</td>
<td>Encode a JSON object UTF-8 buffer into URI parameters</td>
</tr>
<tr>
<td>UrlEncodeJsonObject</td>
<td>Encode a JSON object UTF-8 buffer into URI parameters</td>
</tr>
<tr>
<td>UTF16CharToUtf8</td>
<td>UTF-8 encode one UTF-16 encoded UCS4 character into Dest</td>
</tr>
<tr>
<td>Utf8DecodeToRawUnicode</td>
<td>Convert a UTF-8 string into a RawUnicode string</td>
</tr>
<tr>
<td>Utf8DecodeToRawUnicode</td>
<td>Convert a UTF-8 encoded buffer into a RawUnicode string</td>
</tr>
<tr>
<td>Utf8DecodeToRawUnicodeUI</td>
<td>Convert a UTF-8 string into a RawUnicode string</td>
</tr>
<tr>
<td>Utf8DecodeToRawUnicodeUI</td>
<td>Convert a UTF-8 encoded buffer into a RawUnicode string</td>
</tr>
<tr>
<td>UTF8DecodeToString</td>
<td>Convert any UTF-8 encoded buffer into a generic VCL Text</td>
</tr>
<tr>
<td>UTF8DecodeToString</td>
<td>Convert any UTF-8 encoded buffer into a generic VCL Text</td>
</tr>
<tr>
<td>UTF8FirstLineToUnicodeLength</td>
<td>Calculate the UTF-16 Unicode characters count of the UTF-8 encoded first line</td>
</tr>
<tr>
<td>UTF8IComp</td>
<td>Fast UTF-8 comparison using the NormToUpper[] array for all 8 bits values</td>
</tr>
<tr>
<td>UTF8IComp</td>
<td>Fast UTF-8 comparison using the NormToUpper[] array for all 8 bits values</td>
</tr>
<tr>
<td>UTF8ToInt64</td>
<td>Get the signed 64-bit integer value stored in a RawUTF8 string</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>UTF8ToInteger</td>
<td>Get and check range of a signed 32-bit integer stored in a RawUTF8 string</td>
</tr>
<tr>
<td>UTF8ToInteger</td>
<td>Get the signed 32-bit integer value stored in a RawUTF8 string</td>
</tr>
<tr>
<td>Utf8ToRawUTF8</td>
<td>Direct conversion of a UTF-8 encoded zero terminated buffer into a RawUTF8 String</td>
</tr>
<tr>
<td>UTF8ToShortString</td>
<td>Direct conversion of a UTF-8 encoded buffer into a WinAnsi shortstring buffer</td>
</tr>
<tr>
<td>UTF8ToString</td>
<td>Convert any UTF-8 encoded String into a generic VCL Text</td>
</tr>
<tr>
<td>UTF8ToSynUnicode</td>
<td>Convert any UTF-8 encoded String into a generic SynUnicode Text</td>
</tr>
<tr>
<td>UTF8ToSynUnicode</td>
<td>Convert any UTF-8 encoded String into a generic SynUnicode Text</td>
</tr>
<tr>
<td>UTF8ToSynUnicode</td>
<td>Convert any UTF-8 encoded buffer into a generic SynUnicode Text</td>
</tr>
<tr>
<td>Utf8ToUnicodeLength</td>
<td>Calculate the UTF-16 Unicode characters count, UTF-8 encoded in source^</td>
</tr>
<tr>
<td>UTF8ToWideChar</td>
<td>Convert an UTF-8 encoded text into a WideChar (UTF-16) buffer</td>
</tr>
<tr>
<td>UTF8ToWideChar</td>
<td>Convert an UTF-8 encoded text into a WideChar (UTF-16) buffer</td>
</tr>
<tr>
<td>UTF8ToWideString</td>
<td>Convert any UTF-8 encoded String into a generic WideString Text</td>
</tr>
<tr>
<td>UTF8ToWideString</td>
<td>Convert any UTF-8 encoded String into a generic WideString Text</td>
</tr>
<tr>
<td>Utf8ToWinAnsi</td>
<td>Direct conversion of a UTF-8 encoded zero terminated buffer into a WinAnsi String</td>
</tr>
<tr>
<td>Utf8ToWinAnsi</td>
<td>Direct conversion of a UTF-8 encoded string into a WinAnsi String</td>
</tr>
<tr>
<td>UTF8ToWinPChar</td>
<td>Direct conversion of a UTF-8 encoded buffer into a WinAnsi PAnsiChar buffer</td>
</tr>
<tr>
<td>Utf8TruncatedLength</td>
<td>Compute the truncated length of the supplied UTF-8 value if it exceeds the specified bytes count</td>
</tr>
<tr>
<td>Utf8TruncatedLength</td>
<td>Compute the truncated length of the supplied UTF-8 value if it exceeds the specified bytes count</td>
</tr>
<tr>
<td>Utf8TruncateToLength</td>
<td>Will truncate the supplied UTF-8 value if its length exceeds the specified bytes count</td>
</tr>
<tr>
<td>Utf8TruncateToUnicodeLength</td>
<td>Will truncate the supplied UTF-8 value if its length exceeds the specified UTF-16 Unicode characters count</td>
</tr>
<tr>
<td>UTF8UpperCase</td>
<td>Copy WideChar source into dest^ with upper case conversion, using the NormToUpper[] array for all 8 bits values, encoding the result as UTF-8</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>UTF8UpperCopy255</td>
<td>Copy WideChar source into dest^ with upper case conversion, using the NormToUpper[] array for all 8 bits values, encoding the result as UTF-8</td>
</tr>
<tr>
<td>ValuesToVariantDynArray</td>
<td>Convert an open array list into a dynamic array of variants</td>
</tr>
<tr>
<td>VarDataIsEmptyOrNull</td>
<td>Same as VarIsEmpty(PVariant(V)^) or VarIsEmpty(PVariant(V)^), but faster</td>
</tr>
<tr>
<td>VariantCompare</td>
<td>TVariantCompare-compatible case-sensitive comparison function</td>
</tr>
<tr>
<td>VariantCompareI</td>
<td>TVariantCompare-compatible case-insensitive comparison function</td>
</tr>
<tr>
<td>VariantDynArrayClear</td>
<td>Faster alternative to Finalize(aVariantDynArray)</td>
</tr>
<tr>
<td>VariantDynArrayToJSON</td>
<td>Convert a dynamic array of variants into its JSON serialization</td>
</tr>
<tr>
<td>VariantEquals</td>
<td>Fast comparison of a Variant and UTF-8 encoded String (or number)</td>
</tr>
<tr>
<td>VariantHash</td>
<td>Crc32c-based hash of a variant value</td>
</tr>
<tr>
<td>VariantHexDisplayToBin</td>
<td>Fast conversion from hexa chars, supplied as a variant string, into a binary buffer</td>
</tr>
<tr>
<td>VariantLoad</td>
<td>Retrieve a variant value from our optimized binary serialization format</td>
</tr>
<tr>
<td>VariantLoadJSON</td>
<td>Retrieve a variant value from a JSON number or string</td>
</tr>
<tr>
<td>VariantLoadJSON</td>
<td>Retrieve a variant value from a JSON number or string</td>
</tr>
<tr>
<td>VariantLoadJSON</td>
<td>Retrieve a variant value from a JSON number or string</td>
</tr>
<tr>
<td>VariantSave</td>
<td>Save a Variant content into a destination memory buffer</td>
</tr>
<tr>
<td>VariantSave</td>
<td>Save a Variant content into a binary buffer</td>
</tr>
<tr>
<td>VariantSaveJSON</td>
<td>Save a variant value into a JSON content</td>
</tr>
<tr>
<td>VariantSaveJSON</td>
<td>Save a variant value into a JSON content</td>
</tr>
<tr>
<td>VariantSaveJSONLength</td>
<td>Compute the number of chars needed to save a variant value into a JSON content</td>
</tr>
<tr>
<td>VariantSaveLength</td>
<td>Compute the number of bytes needed to save a Variant content using the VariantSave{} function</td>
</tr>
<tr>
<td>VariantToBoolean</td>
<td>Convert any numericalVariant into a boolean value</td>
</tr>
<tr>
<td>VariantToCurrency</td>
<td>Convert any numericalVariant into a fixed decimals floating point value</td>
</tr>
<tr>
<td>VariantToDateTime</td>
<td>Convert any date/time Variant into a TDateTime value</td>
</tr>
<tr>
<td>VariantToDouble</td>
<td>Convert any numericalVariant into a floating point value</td>
</tr>
<tr>
<td>VariantToDoubleDef</td>
<td>Convert any numerical Variant into a floating point value</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VariantToInlineValue</td>
<td>Convert any Variant into a value encoded as with :(...): inlined parameters in FormatUTF8(Format,Args,Params)</td>
</tr>
<tr>
<td>VariantToInt64</td>
<td>Convert any numerical Variant into a 64-bit integer</td>
</tr>
<tr>
<td>VariantToInt64Def</td>
<td>Convert any numerical Variant into a 64-bit integer</td>
</tr>
<tr>
<td>VariantToInteger</td>
<td>Convert any numerical Variant into a 32-bit integer</td>
</tr>
<tr>
<td>VariantToIntegerDef</td>
<td>Convert any numerical Variant into an integer</td>
</tr>
<tr>
<td>VariantToRawByteString</td>
<td>Convert back a RawByteString from a variant</td>
</tr>
<tr>
<td>VariantToString</td>
<td>Convert any Variant into a VCL string type</td>
</tr>
<tr>
<td>VariantToUTF8</td>
<td>Convert any Variant into UTF-8 encoded String</td>
</tr>
<tr>
<td>VariantToUTF8</td>
<td>Convert any Variant into UTF-8 encoded String</td>
</tr>
<tr>
<td>VariantToUTF8</td>
<td>Convert any Variant into UTF-8 encoded String</td>
</tr>
<tr>
<td>VariantToVariantUTF8</td>
<td>Convert any Variant into another Variant storing an RawUTF8 of the value</td>
</tr>
<tr>
<td>VariantToVarRec</td>
<td>Convert a variant to an open array (const Args: array of const) argument</td>
</tr>
<tr>
<td>VarIs</td>
<td>Allow to check for a specific set of TVarData.VType</td>
</tr>
<tr>
<td>VarIsEmptyOrNull</td>
<td>Same as VarIsEmpty(V) or VarIsEmpty(V), but faster</td>
</tr>
<tr>
<td>VarIs Void</td>
<td>Fastcheck if a variant hold a value</td>
</tr>
<tr>
<td>VarRecAsChar</td>
<td>Get an open array (const Args: array of const) character argument</td>
</tr>
<tr>
<td>VarRecToDouble</td>
<td>Convert an open array (const Args: array of const) argument to a floating point value</td>
</tr>
<tr>
<td>VarRecToInlineValue</td>
<td>Convert an open array (const Args: array of const) argument to a value encoded as with :(...): inlined parameters in FormatUTF8(Format,Args,Params)</td>
</tr>
<tr>
<td>VarRecToInt64</td>
<td>Convert an open array (const Args: array of const) argument to an Int64</td>
</tr>
<tr>
<td>VarRecToTempUTF8</td>
<td>Convert an open array (const Args: array of const) argument to an UTF-8 encoded text, using a specified temporary buffer</td>
</tr>
<tr>
<td>VarRecToUTF8</td>
<td>Convert an open array (const Args: array of const) argument to an UTF-8 encoded text</td>
</tr>
<tr>
<td>VarRecToUTF8IsString</td>
<td>Convert an open array (const Args: array of const) argument to an UTF-8 encoded text, returning FALSE if the argument was not a string value</td>
</tr>
<tr>
<td>VarRecToVariant</td>
<td>Convert an open array (const Args: array of const) argument to a variant</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VarRecToVariant</td>
<td>Convert an open array (const Args: array of const) argument to a variant</td>
</tr>
<tr>
<td>VarStringOrNull</td>
<td>Returns a supplied string as variant, or null if v is void (&quot;&quot;)</td>
</tr>
<tr>
<td>WideStringToUtf8</td>
<td>UTF-8 encode one UTF-16 character into Dest</td>
</tr>
<tr>
<td>WideStringToWinAnsi</td>
<td>Conversion of a wide char into a WinAnsi (CodePage 1252) char index</td>
</tr>
<tr>
<td>WideStringToAnsiChar</td>
<td>Conversion of a wide char into a WinAnsi (CodePage 1252) char</td>
</tr>
<tr>
<td>WideStringToUtf8</td>
<td>Convert a WideString into a UTF-8 string</td>
</tr>
<tr>
<td>WideStringToWinAnsi</td>
<td>Convert a WideString into a WinAnsi (code page 1252) string</td>
</tr>
<tr>
<td>WinAnsiBufferToUtf8</td>
<td>Direct conversion of a WinAnsi PAnsiChar buffer into a UTF-8 encoded buffer</td>
</tr>
<tr>
<td>WinAnsiToRawUnicode</td>
<td>Direct conversion of a WinAnsi (CodePage 1252) string into a Unicode encoded String</td>
</tr>
<tr>
<td>WinAnsiToUnicodeBuffer</td>
<td>Direct conversion of a WinAnsi (CodePage 1252) string into a Unicode buffer</td>
</tr>
<tr>
<td>WinAnsiToUtf8</td>
<td>Direct conversion of a WinAnsi (CodePage 1252) string into a UTF-8 string</td>
</tr>
<tr>
<td>WinAnsiToUtf8</td>
<td>Direct conversion of a WinAnsi (CodePage 1252) string into a UTF-8 string</td>
</tr>
<tr>
<td>WordScanIndex</td>
<td>Fast search of an unsigned Word value position in a Word array</td>
</tr>
<tr>
<td>WriteStringToStream</td>
<td>Write an UTF-8 text into a TStream</td>
</tr>
<tr>
<td>XORMemory</td>
<td>Logical XOR of two memory buffers</td>
</tr>
<tr>
<td>XORMemory</td>
<td>Logical XOR of two memory buffers into a third</td>
</tr>
<tr>
<td>XXHash32</td>
<td>Perform very fast xxHash hashing in 32-bit mode</td>
</tr>
<tr>
<td>YearToPChar</td>
<td>Add the 4 digits of integer Y to P as '0000'..'9999'</td>
</tr>
<tr>
<td>ZeroFill</td>
<td>Same as FillChar(Value^,SizeOf(TVarData),0)</td>
</tr>
<tr>
<td>_Arr</td>
<td>Initialize a variant instance to store some document-based array content</td>
</tr>
<tr>
<td>_ArrFast</td>
<td>Initialize a variant instance to store some document-based array content</td>
</tr>
<tr>
<td>_ByRef</td>
<td>Copy a TDocVariant to another variable, changing the options on the fly</td>
</tr>
<tr>
<td>_ByRef</td>
<td>Copy a TDocVariant to another variable, changing the options on the fly</td>
</tr>
<tr>
<td>_Copy</td>
<td>Return a full nested copy of a document-based variant instance</td>
</tr>
</tbody>
</table>
## Functions or procedures

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>_CopyFast</td>
<td>Return a full nested copy of a document-based variant instance</td>
<td>1070</td>
</tr>
<tr>
<td>_CSV</td>
<td>Convert a TDocVariantData array or a string value into a CSV</td>
<td>1070</td>
</tr>
<tr>
<td>_Json</td>
<td>Initialize a variant instance to store some document-based content from a supplied (extended) JSON content</td>
<td>1071</td>
</tr>
<tr>
<td>_JsonFast</td>
<td>Initialize a variant instance to store some document-based content from a supplied (extended) JSON content</td>
<td>1071</td>
</tr>
<tr>
<td>_JsonFastExt</td>
<td>Initialize a variant instance to store some extended document-based content</td>
<td>1072</td>
</tr>
<tr>
<td>_JsonFastFmt</td>
<td>Initialize a variant instance to store some document-based content from a supplied (extended) JSON content, with parameters formatting</td>
<td>1072</td>
</tr>
<tr>
<td>_JsonFmt</td>
<td>Initialize a variant instance to store some document-based content from a supplied (extended) JSON content, with parameters formatting</td>
<td>1072</td>
</tr>
<tr>
<td>_JsonFmt</td>
<td>Initialize a variant instance to store some document-based content from a supplied (extended) JSON content, with parameters formatting</td>
<td>1072</td>
</tr>
<tr>
<td>_Obj</td>
<td>Initialize a variant instance to store some document-based object content</td>
<td>1073</td>
</tr>
<tr>
<td>_ObjAddProps</td>
<td>Add the property values of a document to a document-based object content</td>
<td>1073</td>
</tr>
<tr>
<td>_ObjAddProps</td>
<td>Add some property values to a document-based object content</td>
<td>1073</td>
</tr>
<tr>
<td>_ObjFast</td>
<td>Initialize a variant instance to store any object as a TDocVariant</td>
<td>1073</td>
</tr>
<tr>
<td>_ObjFast</td>
<td>Initialize a variant instance to store some document-based object content</td>
<td>1073</td>
</tr>
<tr>
<td>_Safe</td>
<td>Direct access to a TDocVariantData from a given variant instance</td>
<td>1074</td>
</tr>
<tr>
<td>_Safe</td>
<td>Delphi has problems inlining this :( direct access to a TDocVariantData from a given variant instance</td>
<td>1074</td>
</tr>
<tr>
<td>_Unique</td>
<td>Ensure a document-based variant instance will have only per-value nested objects or array documents</td>
<td>1074</td>
</tr>
<tr>
<td>_UniqueFast</td>
<td>Ensure a document-based variant instance will have only per-value nested objects or array documents</td>
<td>1074</td>
</tr>
</tbody>
</table>

```plaintext
procedure AddArrayOfConst(var Dest: TVarRecDynArray; const Values: array of const);

Append one or several values to a local "array of const" variable
```
function AddGUID(var guids: TGUIDDynArray; const guid: TGUID; NoDuplicates: boolean=false): integer;
    Append one TGUID item to a TGUID dynamic array
    - returning the newly inserted index in guids[], or an existing index in guids[] if NoDuplicates is TRUE and TGUID already exists

function AddInt64(var Values: TInt64DynArray; var ValuesCount: integer; Value: Int64): PtrInt; overload;
    Add a 64-bit integer value at the end of a dynamic array of integers

function AddInt64(var Values: TInt64DynArray; const Another: TInt64DynArray): PtrInt;
    Add a 64-bit integer array at the end of a dynamic array

function AddInt64(var Values: TInt64DynArray; Value: Int64): PtrInt; overload;
    Add a 64-bit integer value at the end of a dynamic array

function AddInt64Once(var Values: TInt64DynArray; Value: Int64): PtrInt;
    If not already existing, add a 64-bit integer value to a dynamic array

procedure AddInt64Sorted(var Values: TInt64DynArray; Value: Int64);
    If not already existing, add a 64-bit integer value to a sorted dynamic array

procedure AddInteger(var Values: TIntegerDynArray; var ValuesCount: integer; Value: integer); overload;
    Add an integer value at the end of a dynamic array of integers
    - this overloaded function will use a separate Count variable (faster)
    - it won't search for any existing duplicate

function AddInteger(var Values: TIntegerDynArray; Value: integer; NoDuplicates: boolean=false): boolean; overload;
    Add an integer value at the end of a dynamic array of integers
    - returns TRUE if Value was added successfully in Values[], in this case length(Values) will be increased

function AddInteger(var Values: TIntegerDynArray; var ValuesCount: integer; Value: integer; NoDuplicates: boolean): boolean; overload;
    Add an integer value at the end of a dynamic array of integers
    - this overloaded function will use a separate Count variable (faster), and would allow to search for duplicates
    - returns TRUE if Value was added successfully in Values[], in this case ValuesCount will be increased, but length(Values) would stay fixed most of the time (since it stores the Values[] array capacity)

function AddInteger(var Values: TIntegerDynArray; const Another: TIntegerDynArray): PtrInt; overload;
    Add an integer array at the end of a dynamic array of integer

function AddPrefixToCSV(CSV: PUTF8Char; const Prefix: RawUTF8; Sep: AnsiChar = ','): RawUTF8;
    Append some prefix to all CSV values
    AddPrefixToCSV('One,Two,Three','Pre')='PreOne,PreTwo,PreThree'
procedure AddRawUTF8(var Values: TRawUTF8DynArray; var ValuesCount: integer; const Value: RawUTF8); overload;
   Add the Value to Values[], with an external count variable, for performance

function AddRawUTF8(var Values: TRawUTF8DynArray; const Value: RawUTF8; NoDuplicates: boolean=false; CaseSensitive: boolean=true): boolean; overload;
   True if Value was added successfully in Values[]

function AddSortedInteger(var Values: TIntegerDynArray; var ValuesCount: integer; Value: integer; CoValues: PIntegerDynArray=nil): PtrInt; overload;
   Add an integer value in a sorted dynamic array of integers
   - returns the index where the Value was added successfully in Values[]
   - returns -1 if the specified Value was already present in Values[] (we must avoid any duplicate for O(log(n)) binary search)
   - if CoValues is set, its content will be moved to allow inserting a new value at CoValues[result] position

function AddSortedInteger(var Values: TIntegerDynArray; Value: integer; CoValues: PIntegerDynArray=nil): PtrInt; overload;
   Add an integer value in a sorted dynamic array of integers
   - overloaded function which do not expect an external Count variable

function AddSortedRawUTF8(var Values: TRawUTF8DynArray; var ValuesCount: integer; const Value: RawUTF8; CoValues: PIntegerDynArray=nil; ForcedIndex: PtrInt=-1; Compare: TUTF8Compare=nil): PtrInt;
   Add a RawUTF8 value in an alphatically sorted dynamic array of RawUTF8
   - returns the index where the Value was added successfully in Values[]
   - returns -1 if the specified Value was already present in Values[] (we must avoid any duplicate for O(log(n)) binary search)
   - if CoValues is set, its content will be moved to allow inserting a new value at CoValues[result] position - a typical usage of CoValues is to store the corresponding ID to each RawUTF8 item
   - if FastLocatePUTF8CharSorted() has been already called, this index can be set to optional ForcedIndex parameter
   - by default, exact (case-sensitive) match is used; you can specify a custom compare function if needed in Compare optional parameter

procedure AddToCSV(const Value: RawUTF8; var CSV: RawUTF8; const Sep: RawUTF8 = ','); overload;
   Append a Value to a CSV string

function AddWord(var Values: TWordDynArray; var ValuesCount: integer; Value: Word): PtrInt;
   Add a 16-bit integer value at the end of a dynamic array of integers

procedure AndMemory(Dest,Source: PByteArray; size: PtrInt);
   Logical AND of two memory buffers
   - will perform on all buffer bytes:
   Dest[i] := Dest[i] and Source[i];

procedure Ansi7ToString(Text: PWinAnsiChar; Len: PtrInt; var result: string); overload;
   Convert any Ansi 7 bit encoded String into a generic VCL Text
   - the Text content must contain only 7 bit pure ASCII characters
function Ansi7ToString(const Text: RawByteString): string; overload;
  Convert any Ansi 7 bit encoded String into a generic VCL Text
  - the Text content must contain only 7 bit pure ASCII characters

function Ansi7ToString(Text: PWinAnsiChar; Len:_PtrInt): string; overload;
  Convert any Ansi 7 bit encoded String into a generic VCL Text
  - the Text content must contain only 7 bit pure ASCII characters

procedure AnsiCharToUTF8(P: PAnsiChar; L: Integer; var result: RawUTF8; ACP: integer);
  Convert an AnsiChar buffer (of a given code page) into a UTF-8 string

function AnsiIComp(Str1, Str2: pointer): PtrInt;
  Fast WinAnsi comparison using the NormToUpper[] array for all 8 bits values

function AnsiICompW(u1, u2: PWideChar): PtrInt;
  Fast case-insensitive Unicode comparison
  - use the NormToUpperAnsi7Byte[] array, i.e. compare 'a'..'z' as 'A'..'Z'
  - this version expects u1 and u2 to be zero-terminated

procedure AnyAnsiToUTF8(const s: RawByteString; var result: RawUTF8; ACP: integer);
  Direct conversion of an AnsiString with an unknown code page into an UTF-8 encoded String
  - will assume CurrentAnsiConvert.CodePage prior to Delphi 2009
  - newer UNICODE versions of Delphi will retrieve the code page from string

function AnyAnsiToUTF8(const s: RawByteString): RawUTF8; overload;
  Direct conversion of an AnsiString with an unknown code page into an UTF-8 encoded String
  - will assume CurrentAnsiConvert.CodePage prior to Delphi 2009
  - newer UNICODE versions of Delphi will retrieve the code page from string

function AnyScanExists(P, Elem: pointer; Count, ElemSize: PtrInt): boolean;
  Fast search of a binary value position in a fixed-size array
  - Count is the number of entries in P[]

function AnyScanIndex(P, Elem: pointer; Count, ElemSize: PtrInt): PtrInt;
  Fast search of a binary value position in a fixed-size array
  - Count is the number of entries in P[]
  - return index of P^[index]=Elem^, comparing ElemSize bytes
  - return -1 if Value was not found

function AnyTextFileToRawUTF8(const FileName: TFileName; AssumeUTF8IfNoBOM: boolean=false): RawUTF8;
  Get text file contents (even Unicode or UTF8) and convert it into an UTF-8 string according to any
  BOM marker at the beginning of the file
  - if AssumeUTF8IfNoBOM is FALSE, the current string code page is used (i.e. CurrentAnsiConvert class)
  - conversion from ANSI into UTF-8
  - if AssumeUTF8IfNoBOM is TRUE, any file without any BOM marker will be interpreted as UTF-8

function AnyTextFileToString(const FileName: TFileName; ForceUTF8: boolean=false): string;
  Get text file contents (even Unicode or UTF8) and convert it into a Charset-compatible AnsiString
  (for Delphi 7) or an UnicodeString (for Delphi 2009 and up) according to any BOM marker at the
  beginning of the file
  - before Delphi 2009, the current string code page is used (i.e. CurrentAnsiConvert)
function AnyTextFileToSynUnicode(const FileName: TFileName; ForceUTF8: boolean=false): SynUnicode;

Get text file contents (even Unicode or UTF8) and convert it into an Unicode string according to any BOM marker at the beginning of the file
- any file without any BOM marker will be interpreted as plain ASCII: in this case, the current string code page is used (i.e. CurrentAnsiConvert class)

function Append999ToBuffer(Buffer: PUTF8Char; Value: PtrUInt): PUTF8Char;

Fast add text conversion of 0-999 integer value into a given buffer
- warning: it won't check that Value is in 0-999 range
- up to 4 bytes may be written to the buffer (including trailing #0)

procedure AppendBuffersToRawUTF8(var Text: RawUTF8; const Buffers: array of PUTF8Char);

Fast add some characters to a RawUTF8 string
- faster than Text := Text+RawUTF8(Buffers[0])+RawUTF8(Buffers[0])+...

procedure AppendBufferToRawUTF8(var Text: RawUTF8; Buffer: pointer; BufferLen: PtrInt);

Fast add some characters to a RawUTF8 string
- faster than SetString(tmp,Buffer,BufferLen); Text := Text+tmp;

procedure AppendCharToRawUTF8(var Text: RawUTF8; Ch: AnsiChar);

Fast add one character to a RawUTF8 string
- faster than Text := Text + ch;

procedure AppendCSVValues(const CSV: string; const Values: array of string; var Result: string; const AppendBefore: string=#13#10);

Append some text lines with the supplied Values[]
- if any Values[] item is '', no line is added
- otherwise, appends 'Caption: Value', with Caption taken from CSV

function AppendRawUTF8ToBuffer(Buffer: PUTF8Char; const Text: RawUTF8): PUTF8Char;

Fast add some characters from a RawUTF8 string into a given buffer
- warning: the Buffer should contain enough space to store the Text, otherwise you may encounter buffer overflows and random memory errors

procedure AppendShortComma(text: PAnsiChar; len: PtrInt; var result: shortstring; trimlowercase: boolean);

Fast append some UTF-8 text into a shortstring, with an ending ','

procedure AppendToFile(aLine: RawUTF8; const aFileName: TFileName; aMaxSize: Int64=MAXLOGSIZE; aUTCTimeStamp: boolean=false);

Log a message to a local text file
- this version expects the filename to be specified
- format contains the current date and time, then the Msg on one line
- date and time format used is 'YYYYMMDD hh:mm:ss'

function AppendUInt32ToBuffer(Buffer: PUTF8Char; Value: PtrUInt): PUTF8Char;

Fast add text conversion of a 32-bit unsigned integer value into a given buffer
- warning: the Buffer should contain enough space to store the text, otherwise you may encounter buffer overflows and random memory errors
function AreUrlValid(const Url: array of RawUTF8): boolean;
Checks if the supplied UTF-8 text values don't need URI encoding
- returns TRUE if all its chars of all strings are non-void plain ASCII-7 RFC compatible identifiers
  (0..9a..zA..Z_.~)

function ArrayOfConstValueAsText(const NameValuePairs: array of const; const aName: RawUTF8): RawUTF8;
Find a given name in name/value pairs, and returns the value as RawUTF8

function AsciiToBaudot(P: PAnsiChar; len:_PTRInt): RawByteString; overload;
Convert some ASCII-7 text into binary, using Emile Baudot code
- as used in telegraphs, covering #10 #13 #32 a-z 0-9-’,!:(+) $ ? @ . / ; charset, following a
  custom static-huffman-like encoding with 5-bit masks
- any upper case char will be converted into lowercase during encoding
- other characters (e.g. UTF-8 accents, or controls chars) will be ignored
- resulting binary will consume 5 (or 10) bits per character
- reverse of the BaudotToAscii() function
- the "baud" symbol rate measurement comes from Emile's name ;)

function AsciiToBaudot(const Text: RawUTF8): RawByteString; overload;
Convert some ASCII-7 text into binary, using Emile Baudot code
- as used in telegraphs, covering #10 #13 #32 a-z 0-9-’,!:(+) $ ? @ . / ; charset, following a
  custom static-huffman-like encoding with 5-bit masks
- any upper case char will be converted into lowercase during encoding
- other characters (e.g. UTF-8 accents, or controls chars) will be ignored
- resulting binary will consume 5 (or 10) bits per character
- reverse of the BaudotToAscii() function
- the "baud" symbol rate measurement comes from Emile's name ;)

function Base64Decode(sp,rp: PAnsiChar; len: PTRInt): boolean;
Direct low-level decoding of a Base64 encoded buffer
- here len is the number of 4 chars chunks in sp input
- deprecated low-level function: use Base64ToBin/Base64ToBinSafe instead

function Base64MagicCheckAndDecode(Value: PUTF8Char; var Blob: RawByteString): boolean; overload;
Check and decode \uFFF0base64encodedbinary' content into binary
- this method will check the supplied value to match the expected JSON_BASE64_MAGIC pattern,
  decode and set Blob and return TRUE

function Base64MagicCheckAndDecode(Value: PUTF8Char; ValueLen: Integer; var Blob: RawByteString): boolean; overload;
Check and decode \uFFF0base64encodedbinary' content into binary
- this method will check the supplied value to match the expected JSON_BASE64_MAGIC pattern,
  decode and set Blob and return TRUE

function Base64MagicCheckAndDecode(Value: PUTF8Char; var Blob: TSynTempBuffer): boolean; overload;
Check and decode \uFFF0base64encodedbinary' content into binary
- this method will check the supplied value to match the expected JSON_BASE64_MAGIC pattern,
  decode and set Blob and return TRUE
procedure Base64MagicDecode(var ParamValue: RawUTF8);

Just a wrapper around Base64ToBin() for in-place decode of JSON_BASE64_MAGIC
\"uFFF0base64encodedbinary\" content into binary
- input ParamValue shall have been checked to match the expected pattern

function Base64ToBin(sp: PAnsiChar; len: PtrInt; var data: RawByteString): boolean; overload;
  Fast conversion from Base64 encoded text into binary data
  - is now just an alias to Base64ToBinSafe() overloaded function
  - returns false and data="" if sp/len buffer was invalid

function Base64ToBin(sp: PAnsiChar; len: PtrInt): RawByteString; overload;
  Fast conversion from Base64 encoded text into binary data
  - is now just an alias to Base64ToBinSafe() overloaded function
  - returns "" if sp/len buffer was not a valid Base64-encoded input

function Base64ToBin(const s: RawByteString): RawByteString; overload;
  Fast conversion from Base64 encoded text into binary data
  - is now just an alias to Base64ToBinSafe() overloaded function
  - returns "" if s was not a valid Base64-encoded input

function Base64ToBin(const base64: RawByteString; bin: PAnsiChar; binlen: PtrInt; nofullcheck: boolean=true): boolean; overload;
  Fast conversion from Base64 encoded text into binary data
  - returns TRUE on success, FALSE if base64 does not match binlen
  - nofullcheck is deprecated and not used any more, since nofullcheck=false is now processed with
  no performance cost

function Base64ToBin(base64, bin: PAnsiChar; base64len, binlen: PtrInt; nofullcheck: boolean=true): boolean; overload;
  Fast conversion from Base64 encoded text into binary data
  - returns TRUE on success, FALSE if base64 does not match binlen
  - nofullcheck is deprecated and not used any more, since nofullcheck=false is now processed with
  no performance cost

function Base64ToBin(sp: PAnsiChar; len: PtrInt; var Blob: TSynTempBuffer): boolean; overload;
  Fast conversion from Base64 encoded text into binary data
  - returns TRUE on success, FALSE if sp/len buffer was invalid

function Base64ToBinLength(sp: PAnsiChar; len: PtrInt): PtrInt;
  Retrieve the expected undecoded length of a Base64 encoded buffer
  - here len is the number of bytes in sp

function Base64ToBinLengthSafe(sp: PAnsiChar; len: PtrInt): PtrInt;
  Retrieve the expected undecoded length of a Base64 encoded buffer
  - here len is the number of bytes in sp
  - will check supplied text is a valid Base64 encoded stream

function Base64ToBinSafe(sp: PAnsiChar; len: PtrInt; var data: RawByteString): boolean; overload;
  Fast conversion from Base64 encoded text into binary data
  - will check supplied text is a valid Base64 encoded stream
function Base64ToBinSafe(const s: RawByteString): RawByteString; overload;

*Fast conversion from Base64 encoded text into binary data*
- will check supplied text is a valid Base64 encoded stream

function Base64ToBinSafe(sp: PAnsiChar; len: PtrInt): RawByteString; overload;

*Fast conversion from Base64 encoded text into binary data*
- will check supplied text is a valid Base64 encoded stream

procedure Base64ToURI(var base64: RawUTF8);

*Conversion from any Base64 encoded value into URI-compatible encoded text*
- warning: will modify the supplied base64 string in-place
- in comparison to Base64 standard encoding, will trim any right-sided '=' unsignificant characters, and replace '+' or '/' by '_' or '.'

function Base64uriDecode(sp,rp: PAnsiChar; len: PtrInt): boolean;

*Direct low-level decoding of a Base64-URI encoded buffer*
- the buffer is expected to be at least Base64uriToBinLength() bytes long
- returns true if the supplied sp[] buffer has been successfully decoded into rp[] - will break at any invalid character, so is always safe to use
- in comparison to Base64 standard encoding, will trim any right-sided '=' unsignificant characters, and replace '+' or '/' by '_' or '.'
- you should better not use this, but Base64uriToBin() overloaded functions

procedure Base64uriEncode(rp, sp: PAnsiChar; len: cardinal);

*Low-level conversion from a binary buffer into Base64-like URI-compatible encoded text*
- you should rather use the overloaded BinToBase64uri() functions

function Base64uriToBin(const s: RawByteString): RawByteString; overload;

*Fast conversion from Base64-URI encoded text into binary data*
- in comparison to Base64 standard encoding, will trim any right-sided '=' unsignificant characters, and replace '+' or '/' by '_' or '.'

function Base64uriToBin(sp: PAnsiChar; len: PtrInt; var temp: TSynTempBuffer): boolean; overload;

*Fast conversion from Base64-URI encoded text into binary data*
- caller should always execute temp.Done when finished with the data
- in comparison to Base64 standard encoding, will trim any right-sided '=' unsignificant characters, and replace '+' or '/' by '_' or '.'

function Base64uriToBin(const base64: RawByteString; bin: PAnsiChar; binlen: PtrInt): boolean; overload;

*Fast conversion from Base64-URI encoded text into binary data*
- in comparison to Base64 standard encoding, will trim any right-sided '=' unsignificant characters, and replace '+' or '/' by '_' or '.'
- will check supplied text is a valid Base64-URI encoded stream

function Base64uriToBin(base64, bin: PAnsiChar; base64len, binlen: PtrInt): boolean; overload;

*Fast conversion from Base64-URI encoded text into binary data*
- in comparison to Base64 standard encoding, will trim any right-sided '=' unsignificant characters, and replace '+' or '/' by '_' or '.'
- will check supplied text is a valid Base64-URI encoded stream
**function** Base64uriToBin(sp: PAnsiChar; len:.PtrInt): RawByteString; overload;

*Fast conversion from Base64-URI encoded text into binary data*
- in comparison to Base64 standard encoding, will trim any right-sided '=' unsignificant characters, and replace '+' or '/' by '_' or '.'

**procedure** Base64uriToBin(sp: PAnsiChar; len:.PtrInt; var result: RawByteString); overload;

*Fast conversion from Base64-URI encoded text into binary data*
- in comparison to Base64 standard encoding, will trim any right-sided '=' unsignificant characters, and replace '+' or '/' by '_' or '.'

**function** Base64uriToBinLength(len: PtrInt): PtrInt;

*Retrieve the expected undecoded length of a Base64-URI encoded buffer*
- here len is the number of bytes in sp
- in comparison to Base64 standard encoding, will trim any right-sided '=' unsignificant characters, and replace '+' or '/' by '_' or '.'

**function** BaudotToAscii(Baudot: PByteArray; len: PtrInt): RawUTF8; overload;

*Convert some Baudot code binary, into ASCII-7 text*
- reverse of the AsciiToBaudot() function
- any uppercase character would be decoded as lowercase - and some characters may have disappeared
- the "baud" symbol rate measurement comes from Emile's name ;)

**function** BaudotToAscii(const Baudot: RawByteString): RawUTF8; overload;

*Convert some Baudot code binary, into ASCII-7 text*
- reverse of the AsciiToBaudot() function
- any uppercase character would be decoded as lowercase - and some characters may have disappeared
- the "baud" symbol rate measurement comes from Emile's name ;)

**function** BinToBase64(const data, Prefix, Suffix: RawByteString; WithMagic: boolean): RawUTF8; overload;

*Fast conversion from binary data into prefixed/suffixed Base64 encoded UTF-8 text*
- with optional JSON_BASE64_MAGIC prefix (UTF-8 encoded \uFFF0 special code)

**function** BinToBase64(Bin: PAnsiChar; BinBytes: integer): RawUTF8; overload;

*Fast conversion from binary data into Base64 encoded UTF-8 text*

**function** BinToBase64(const s: RawByteString): RawUTF8; overload;

*Fast conversion from binary data into Base64 encoded UTF-8 text*

**function** BinToBase64Length(len: PtrUInt): PtrUInt;

*Retrieve the expected encoded length after Base64 process*

**function** BinToBase64Short(const s: RawByteString): shortstring; overload;

*Fast conversion from a small binary data into Base64 encoded UTF-8 text*

**function** BinToBase64Short(Bin: PAnsiChar; BinBytes: integer): shortstring; overload;

*Fast conversion from a small binary data into Base64 encoded UTF-8 text*
function BinToBase64uri(Bin: PAnsiChar; BinBytes: integer): RawUTF8; overload;
    "Fast conversion from a binary buffer into Base64-like URI-compatible encoded text"
    - in comparison to Base64 standard encoding, will trim any right-sided '=' unsignificant characters,
      and replace '+' or '/' by '_' or '.'

function BinToBase64uri(const s: RawByteString): RawUTF8; overload;
    "Fast conversion from binary data into Base64-like URI-compatible encoded text"
    - in comparison to Base64 standard encoding, will trim any right-sided '=' unsignificant characters,
      and replace '+' or '/' by '_' or '.'

function BinToBase64uriLength(len: PtrUInt): PtrUInt;
    "Retrieve the expected encoded length after Base64-URI process"
    - in comparison to Base64 standard encoding, will trim any right-sided '=' unsignificant characters,
      and replace '+' or '/' by '_' or '.'

function BinToBase64uriShort(Bin: PAnsiChar; BinBytes: integer): shortstring;
    "Fast conversion from a binary buffer into Base64-like URI-compatible encoded shortstring"
    - in comparison to Base64 standard encoding, will trim any right-sided '=' unsignificant characters,
      and replace '+' or '/' by '_' or '.'
    - returns '' if BinBytes void or too big for the resulting shortstring

function BinToBase64WithMagic(Data: pointer; DataLen: integer): RawUTF8; overload;
    "Fast conversion from binary data into Base64 encoded UTF-8 text with JSON_BASE64_MAGIC prefix (UTF-8 encoded ￰ special code)"

function BinToBase64WithMagic(const data: RawByteString): RawUTF8; overload;
    "Fast conversion from binary data into Base64 encoded UTF-8 text with JSON_BASE64_MAGIC prefix (UTF-8 encoded ￰ special code)"

function BinToHex(Bin: PAnsiChar; BinBytes: integer): RawUTF8; overload;
    "Fast conversion from binary data into hexa chars"

procedure BinToHex(Bin, Hex: PAnsiChar; BinBytes: integer); overload;
    "Fast conversion from binary data into hexa chars"
    - BinBytes contain the bytes count to be converted: Hex^ must contain enough space for at least
      BinBytes*2 chars
    - using this function with BinBytes^ as an integer value will encode it in low-endian order
      (less-significant byte first): don't use it for display

function BinToHex(const Bin: RawByteString): RawUTF8; overload;
    "Fast conversion from binary data into hexa chars"

procedure BinToHexDisplay(Bin, Hex: PAnsiChar; BinBytes: integer); overload;
    "Fast conversion from binary data into hexa chars, ready to be displayed"
    - BinBytes contain the bytes count to be converted: Hex^ must contain enough space for at least
      BinBytes*2 chars
    - using this function with Bin^ as an integer value will encode it in big-endian order
      (most-significant byte first): use it for display

function BinToHexDisplay(Bin: PAnsiChar; BinBytes: integer): RawUTF8; overload;
    "Fast conversion from binary data into hexa chars, ready to be displayed"
**function** BinToHexDisplayFile(Bin: PAnsiChar; BinBytes: integer): TFileName;

*Fast conversion from binary data into hexa lowercase chars, ready to be used as a convenient TFileName prefix*

**procedure** BinToHexDisplayLower(Bin, Hex: PAnsiChar; BinBytes: PtrInt); overload;

*Fast conversion from binary data into lowercase hexa chars*
- BinBytes contain the bytes count to be converted: Hex^ must contain enough space for at least BinBytes*2 chars
- using this function with Bin^ as an integer value will encode it in big-endian order (most-significant byte first): use it for display

**function** BinToHexDisplayLower(Bin: PAnsiChar; BinBytes: integer): RawUTF8; overload;

*Fast conversion from binary data into lowercase hexa chars*

**function** BinToHexDisplayLowerShort(Bin: PAnsiChar; BinBytes: integer): shortstring;

*Fast conversion from up to 127 bytes of binary data into lowercase hexa chars*

**function** BinToHexDisplayLowerShort16(Bin: Int64; BinBytes: integer): TShort16;

*Fast conversion from up to 64-bit of binary data into lowercase hexa chars*

**function** BinToHexDisplayLowerVariant(Bin: pointer; BinBytes: integer): variant;

*Fast conversion of a binary buffer into hexa chars, as a variant string*

**procedure** BinToHexLower(Bin, Hex: PAnsiChar; BinBytes: integer); overload;

*Fast conversion from binary data into lowercase hexa chars*
- BinBytes contain the bytes count to be converted: Hex^ must contain enough space for at least BinBytes*2 chars
- using this function with BinBytes^ as an integer value will encode it in low-endian order (less-significant byte first): don't use it for display

**procedure** BinToHexLower(Bin: PAnsiChar; BinBytes: integer; var result: RawUTF8); overload;

*Fast conversion from binary data into lowercase hexa chars*

**function** BinToHexLower(Bin: PAnsiChar; BinBytes: integer): RawUTF8; overload;

*Fast conversion from binary data into lowercase hexa chars*

**function** BinToHexLower(const Bin: RawByteString): RawUTF8; overload;

*Fast conversion from binary data into lowercase hexa chars*

**function** BinToSource(const ConstName, Comment: RawUTF8; Data: pointer; Len: integer; PerLine: Integer=16; const Suffix: RawUTF8=''): RawUTF8; overload;

*Generate some pascal source code holding some data binary as constant*
- can store sensitive information (e.g. certificates) within the executable
- generates a source code snippet of the following format:

```pascal
const
// Comment
ConstName: array[0..2] of byte = ($01,$02,$03);
```
procedure BinToSource(Dest: TTextWriter; const ConstName, Comment: RawUTF8; Data: pointer; Len: integer; PerLine: integer=16); overload;

Generate some pascal source code holding some data binary as constant
- can store sensitive information (e.g. certificates) within the executable
- generates a source code snippet of the following format:

```
const
  // Comment
  ConstName: array[0..2] of byte = ( // 01, 02, 03);
```

function bswap32(a: cardinal): cardinal;

Convert the endianness of a given unsigned 32-bit integer into BigEndian

function bswap64(const a: QWord): QWord;

Convert the endianness of a given unsigned 64-bit integer into BigEndian

procedure bswap64array(a,b: PQWordArray; n: PtrInt);

Convert the endianness of an array of unsigned 64-bit integer into BigEndian
- n is required to be > 0
- warning: on x86, a should be <> b

function BufferLineLength(Text, TextEnd: PUTF8Char): PtrInt;

Compute the line length from a size-delimited source array of chars
- will use fast assembly on x86-64 CPU, and expects TextEnd to be not nil
- is likely to read some bytes after the TextEnd buffer, so GetLineSize() may be preferred, e.g. on memory mapped files

function ByteScanIndex(P: PByteArray; Count: PtrInt; Value: Byte): PtrInt;

Fast search of an unsigned Byte value position in a Byte array
- Count is the number of Byte entries in P^ 
- return index of P^[index]=Value
- return -1 if Value was not found

procedure BytesToRawByteString(const bytes: TBytes; out buf: RawByteString);

Creates a RawByteString memory buffer from a TBytes content

function ByteToHex(P: PAnsiChar; Value: byte): PAnsiChar;

Append one byte as hexadecimal char pairs, into a text buffer

procedure CamelCase(const text: RawUTF8; var s: RawUTF8; const isWord: TSynByteSet=[ord('0')..ord('9'),ord('a')..ord('z'),ord('A')..ord('Z')]); overload;

Convert a string into an human-friendly CamelCase identifier
- replacing spaces or punctuations by an uppercase character
- as such, it is not the reverse function to UnCamelCase()

procedure CamelCase(P: PAnsiChar; len: PIntPtr; var s: RawUTF8; const isWord: TSynByteSet=[ord('0')..ord('9'),ord('a')..ord('z'),ord('A')..ord('Z')]); overload;

Convert a string into an human-friendly CamelCase identifier
- replacing spaces or punctuations by an uppercase character
- as such, it is not the reverse function to UnCamelCase()
**function** CardinalToHex(aCardinal: Cardinal): RawUTF8;

*Fast conversion from a Cardinal value into hexa chars, ready to be displayed*
- use internally BinToHexDisplay()
- reverse function of HexDisplayToCardinal()

**function** CardinalToHexLower(aCardinal: Cardinal): RawUTF8;

*Fast conversion from a Cardinal value into hexa chars, ready to be displayed*
- use internally BinToHexDisplayLower()
- reverse function of HexDisplayToCardinal()

**function** CardinalToHexShort(aCardinal: Cardinal): TShort16;

*Fast conversion from a Cardinal value into hexa chars, ready to be displayed*
- use internally BinToHexDisplay()
- such result type would avoid a string allocation on heap

**function** Char2ToByte(P: PUTF8Char; out Value: Cardinal): Boolean;

*Fast conversion of 2 digit characters into a 0..99 value*
- returns FALSE on success, TRUE if P^ is not correct

**function** Char3ToWord(P: PUTF8Char; out Value: Cardinal): Boolean;

*Fast conversion of 3 digit characters into a 0..9999 value*
- returns FALSE on success, TRUE if P^ is not correct

**function** Char4ToWord(P: PUTF8Char; out Value: Cardinal): Boolean;

*Fast conversion of 4 digit characters into a 0..9999 value*
- returns FALSE on success, TRUE if P^ is not correct

**function** Chars3ToInt18(P: pointer): cardinal;

*Revert the value as encoded by TTextWriter.AddInt18ToChars3() or Int18ToChars3()*
- no range check is performed: you should ensure that the incoming text follows the expected 3-chars layout

**function** CharSetToCodePage(CharSet: integer): cardinal;

*Convert a char set to a code page*

**function** ClassNameShort(Instance: TObject): PShortString; overload;

*Just a wrapper around vmtClassName to avoid a string conversion*

**function** ClassNameShort(C: TClass): PShortString; overload;

*Just a wrapper around vmtClassName to avoid a string conversion*

**function** CodePageToCharSet(CodePage: Cardinal): Integer;

*Convert a code page to a char set*

**function** CompareCardinal(const A, B: cardinal): integer;

*A comparison function for sorting 32-bit unsigned integer values*

**function** CompareFloat(const A, B: double): integer;

*A comparison function for sorting IEEE 754 double precision values*

**function** CompareInt64(const A, B: Int64): integer;

*A comparison function for sorting 64-bit signed integer values*

**function** CompareInteger(const A, B: integer): integer;

*A comparison function for sorting 32-bit signed integer values*
function CompareMem(P1, P2: Pointer; Length: PtrInt): Boolean;
  *Our fast version of CompareMem() with optimized asm for x86 and tune pascal*

function CompareMemSmall(P1, P2: Pointer; Length: PtrUInt): Boolean;
  A CompareMem()-like function designed for small (a few bytes) content

function CompareQWord(A, B: QWord): integer;
  A comparison function for sorting 64-bit unsigned integer values
  - note that QWord(A)>QWord(B) is wrong on older versions of Delphi, so you should better use
    this function or SortDynArrayQWord() to properly compare two QWord values over CPUX86

function CompressSynLZ(var DataRawByteString; Compress: boolean): AnsiString;
  Compress a data content using the SynLZ algorithm
  - as expected by THttpSocket.RegisterCompress
  - will return 'synlz' as ACCEPT-ENCODING: header parameter
  - will store a hash of both compressed and uncompressed stream: if the data is corrupted during
    transmission, will instantly return "

function ContainsUTF8(p, up: PUTF8Char): boolean;
  Return true if up\(^{^}\) is contained inside the UTF-8 buffer p\(^{\text{^}}\)
  - search up\(^{\text{^}}\) at the beginning of every UTF-8 word (aka in Soundex)
  - here a "word" is a Win-Ansi word, i.e. '0'..'9', 'A'..'Z'
  - up\(^{\text{^}}\) must be already Upper

function ConvertCaseUTF8(P: PUTF8Char; const Table: TNormTableByte): PtrInt;
  Fast conversion of the supplied text into 8 bit case sensitivity
  - convert the text in-place, returns the resulting length
  - it will decode the supplied UTF-8 content to handle more than 7 bit of ascii characters during the
    conversion (leaving not WinAnsi characters untouched)
  - will not set the last char to #0 (caller must do that if necessary)

procedure CopyAndSortInt64(Values: PInt64Array; ValuesCount: integer; var Dest: TInt64DynArray);
  Copy an integer array, then sort it, low values first

procedure CopyAndSortInteger(Values: PIntegerArray; ValuesCount: integer; var Dest: TIntegerDynArray);
  Copy an integer array, then sort it, low values first

function CopyFile(const Source, Target: TFileName; FailIfExists: boolean): boolean;
  Copy one file to another, similar to the Windows API

procedure CopyInt64(const Source: TInt64DynArray; out Dest: TInt64DynArray);
  Create a new 64-bit integer dynamic array with the values from another one

procedure CopyInteger(const Source: TIntegerDynArray; out Dest: TIntegerDynArray);
  Create a new 32-bit integer dynamic array with the values from another one

procedure crc128c(buf: PAnsiChar; len: cardinal; out crc: THash128);
  Compute a 128-bit checksum on the supplied buffer, cascading two crc32c
  - will use SSE 4.2 hardware accelerated instruction, if available
  - will combine two crc32c() calls into a single TAESBlock result
  - by design, such combined hashes cannot be cascaded
function crc16(Data: PAnsiChar; Len: integer): cardinal;

Compute CRC16-CCITT checkum on the supplied buffer
- i.e. 16-bit CRC-CCITT, with polynomial $x^{16} + x^{12} + x^5 + 1$ ($1021$) and $\$ffff$ as initial value
- this version is not optimized for speed, but for correctness

procedure crc256c(buf: PAnsiChar; len: cardinal; out crc: THash256);

Compute a 256-bit checksum on the supplied buffer using crc32c
- will use SSE 4.2 hardware accelerated instruction, if available
- will combine two crc32c() calls into a single THash256 result
- by design, such combined hashes cannot be cascaded

function crc32cfast(crc: cardinal; buf: PAnsiChar; len: cardinal): cardinal;

Compute CRC32C checksum on the supplied buffer on processor-neutral code
- result is compatible with SSE 4.2 based hardware accelerated instruction
- will use fast x86/x64 asm or efficient pure pascal implementation on ARM
- result is not compatible with zlib's crc32() - not the same polynomial
- crc32cfast() is 1.7 GB/s, crc32csse42() is 4.3 GB/s
- you should use crc32c() function instead of crc32cfast() or crc32csse42()

function crc32cinlined(crc: cardinal; buf: PAnsiChar; len: cardinal): cardinal;

Compute CRC32C checksum on the supplied buffer using inlined code
- if the compiler supports inlining, will compute a slow but safe crc32c checksum of the binary buffer, without calling the main crc32c() function
- may be used e.g. to identify patched executable at runtime, for a licensing protection system

function crc32csse42(crc: cardinal; buf: PAnsiChar; len: cardinal): cardinal;

Compute CRC32C checksum on the supplied buffer using SSE 4.2
- use Intel Streaming SIMD Extensions 4.2 hardware accelerated instruction
- SSE 4.2 shall be available on the processor (i.e. cfSSE42 in CpuFeatures)
- result is not compatible with zlib's crc32() - not the same polynomial
- crc32cfast() is 1.7 GB/s, crc32csse42() is 4.3 GB/s
- you should use crc32c() function instead of crc32cfast() or crc32csse42()

function crc32cUTF8ToHex(const str: RawUTF8): RawUTF8;

Compute the hexadecimal representation of the crc32 checksum of a given text
- wrapper around CardinalToHex(crc32c(...))

procedure crc512c(buf: PAnsiChar; len: cardinal; out crc: THash512);

Compute a 512-bit checksum on the supplied buffer using crc32c
- will use SSE 4.2 hardware accelerated instruction, if available
- will combine two crc32c() calls into a single THash512 result
- by design, such combined hashes cannot be cascaded

function crc63c(buf: PAnsiChar; len: cardinal): Int64;

Compute CRC63C checksum on the supplied buffer, cascading two crc32c
- similar to crc64c, but with 63-bit, so no negative value: may be used safely e.g. as mORMot's TID source
- will use SSE 4.2 hardware accelerated instruction, if available
- will combine two crc32c() calls into a single Int64 result
- by design, such combined hashes cannot be cascaded
function crc64c(buf: PAnsiChar; len: cardinal): Int64;

Compute CRC64C checksum on the supplied buffer, cascading two crc32c
- will use SSE 4.2 hardware accelerated instruction, if available
- will combine two crc32c() calls into a single Int64 result
- by design, such combined hashes cannot be cascaded

procedure crcblockNoSSE42(crc128, data128: PBlock128);

Computation of our 128-bit CRC of a 128-bit binary buffer without SSE4.2
- to be used for regression tests only: crcblock will use the fastest implementation available on the current CPU (e.g. with SSE 4.2 opcodes)

procedure crcblocksfast(crc128, data128: PBlock128; count: integer);

Compute a proprietary 128-bit CRC of 128-bit binary buffers
- to be used for regression tests only: crcblocks will use the fastest implementation available on the current CPU (e.g. with SSE 4.2 opcodes)

function CreateInternalWindow(const aWindowName: string; aObject: TObject): HWND;

This function can be used to create a GDI compatible window, able to receive Windows Messages for fast local communication
- will return 0 on failure (window name already existing e.g.), or the created HWND handle on success
- it will call the supplied message handler defined for a given Windows Message: for instance, define such a method in any object definition:

procedure WMCopyData(var Msg : TWMCopyData); message WM_COPYDATA;

function CSVEncode(const NameValuePairs: array of const; const KeySeparator: RawUTF8='='; const ValueSeparator: RawUTF8='#13#10'): RawUTF8;

Encode name/value pairs into CSV/INI raw format

function CSVOfValue(const Value: RawUTF8; Count: cardinal; const Sep: RawUTF8=','): RawUTF8;

Return a CSV list of the iterated same value
- e.g. CSVOfValue('?',3)='?,?,?'

function CSVToInt64DynArray(CSV: PUTF8Char; Sep: AnsiChar=','): TInt64DynArray;

Convert the strings in the specified CSV text into a dynamic array of integer

procedure CSVToInt64DynArray(CSV: PUTF8Char; var Result: TInt64DynArray; Sep:AnsiChar=','); overload;

Append the strings in the specified CSV text into a dynamic array of integer

procedure CSVToIntegerDynArray(CSV: PUTF8Char; var Result: TIntegerDynArray; Sep: AnsiChar=','); overload;

Append the strings in the specified CSV text into a dynamic array of integer

procedure CSVToRawUTF8DynArray(const CSV,Sep,SepEnd: RawUTF8; var Result: TRawUTF8DynArray); overload;

Add the strings in the specified CSV text into a dynamic array of UTF-8 strings

procedure CSVToRawUTF8DynArray(CSV: PUTF8Char; var Result: TRawUTF8DynArray; Sep: AnsiChar=';', TrimItems: boolean=false; AddVoidItems: boolean=false); overload;

Add the strings in the specified CSV text into a dynamic array of UTF-8 strings
function Curr64ToPChar(const Value: Int64; Dest: PUTF8Char): PtrInt;

*Convert an INTEGER Curr64 (value*10000) into a string*
- this type is compatible with Delphi currency memory map with PInt64(@Curr)^
- fast conversion, using only integer operations
- decimals are joined by 2 (no decimal, 2 decimals, 4 decimals)
- return the number of chars written to Dest^

function Curr64ToStr(const Value: Int64): RawUTF8; overload;

*Convert an INTEGER Curr64 (value*10000) into a string*
- this type is compatible with Delphi currency memory map with PInt64(@Curr)^
- fast conversion, using only integer operations
- decimals are joined by 2 (no decimal, 2 decimals, 4 decimals)

procedure Curr64ToStr(const Value: Int64; var result: RawUTF8); overload;

*Convert an INTEGER Curr64 (value*10000) into a string*
- this type is compatible with Delphi currency memory map with PInt64(@Curr)^
- fast conversion, using only integer operations
- decimals are joined by 2 (no decimal, 2 decimals, 4 decimals)

function Curr64ToString(Value: Int64): string;

*Convert a currency value from its Int64 binary representation into its numerical text equivalency*
- decimals are joined by 2 (no decimal, 2 decimals, 4 decimals)

function CurrencyToStr(Value: currency): RawUTF8;

*Convert a currency value into a string*
- fast conversion, using only integer operations
- decimals are joined by 2 (no decimal, 2 decimals, 4 decimals)

function DateTimeMSToString(DateTime: TDateTime; Expanded: boolean=true; FirstTimeChar: AnsiChar=' '; const TZD: RawUTF8='Z'): RawUTF8; overload;

*Convert some date/time to the ISO 8601 text layout, including milliseconds*
- i.e. 'YYYY-MM-DD hh:mm:ss.sssZ' or 'YYYYMMDD hhmmss.sssZ' format
- TZD is the ending time zone designator ('', 'Z' or '+hh:mm' or '-hh:mm')
- see also TTextWriter.AddDateTimeMS method

function DateTimeMSToString(HH, MM, SS, MS, Y, M, D: cardinal; Expanded: boolean; FirstTimeChar: AnsiChar=' '; const TZD: RawUTF8='Z'): RawUTF8; overload;

*Convert some date/time to the ISO 8601 text layout, including milliseconds*
- i.e. 'YYYY-MM-DD hh:mm:ss.sssZ' or 'YYYYMMDD hhmmss.sssZ' format
- TZD is the ending time zone designator ('', 'Z' or '+hh:mm' or '-hh:mm')
- see also TTextWriter.AddDateTimeMS method

procedure DateTimeToFileShort(const DateTime: TDateTime; out result: TShort16); overload;

*Delphi 2007 is buggy as hell convert some TDateTime to a small text layout, perfect e.g. for naming a local file*
- use 'YYYYMMDDHHMMSS' format so year is truncated to last 2 digits, expecting a date > 1999 (a current date would be fine)

function DateTimeToFileShort(const DateTime: TDateTime): TShort16; overload;

*Convert some TDateTime to a small text layout, perfect e.g. for naming a local file*
- use 'YYYYMMDDHHMMSS' format so year is truncated to last 2 digits, expecting a date > 1999 (a current date would be fine)
function DateTimeToHTTPDate(dt: TDateTime; const tz: RawUTF8='GMT'): RawUTF8;
overload;

Convert some date/time to the "HTTP-date" format as defined by RFC 7231
- i.e. "Tue, 15 Nov 1994 12:45:26 GMT" to be used as a value of "Date", "Expires" or
"Last-Modified" HTTP header
- if you care about timezones Value must be converted to UTC first using
TSynTimeZone.LocalToUtc, or tz should be properly set

function DateTimeToi18n(const DateTime: TDateTime): string;
Wrapper calling global i18nDateTimeText() callback if set, or returning ISO-8601 standard layout
on default

function DateTimeToIso8601(D: TDateTime; Expanded: boolean; FirstChar: AnsiChar='T'; WithMS: boolean=false; QuotedChar: AnsiChar=#0): RawUTF8;
overload;

Basic Date/Time conversion into ISO-8601
- use 'YYYY-MM-DDThhmmss' format if not Expanded
- use 'YYYY-MM-DDThh:mm:ss' format if Expanded
- if WithMS is TRUE, will append '.sss' for milliseconds resolution
- if QuotedChar is not default #0, will (double) quote the resulted text
- you may rather use DateTimeToIso8601Text() to handle 0 or date-only values

function DateTimeToIso8601(P: PUTF8Char; D: TDateTime; Expanded: boolean; FirstChar: AnsiChar='T'; WithMS: boolean=false; QuotedChar: AnsiChar=#0): integer;
overload;

Basic Date/Time conversion into ISO-8601
- use 'YYYY-MM-DDThhmmss' format if not Expanded
- use 'YYYY-MM-DDThh:mm:ss' format if Expanded
- if WithMS is TRUE, will append '.sss' for milliseconds resolution
- if QuotedChar is not default #0, will (double) quote the resulted text
- you may rather use DateTimeToIso8601Text() to handle 0 or date-only values
- returns the number of chars written to P^ buffer

function DateTimeToIso8601ExpandedPChar(const Value: TDateTime; Dest: PUTF8Char; FirstChar: AnsiChar='T'; WithMS: boolean=false): PUTF8Char;
Write a TDateTime value, expanded as Iso-8601 encoded text into P^ Ansi buffer
- if DT=0, returns ''
- if DT contains only a date, returns the date encoded as 'YYYY-MM-DD'
- if DT contains only a time, returns the time encoded as 'Thh:mm:ss'
- otherwise, returns the ISO-8601 date and time encoded as 'YYYY-MM-DDThh:mm:ss'
- if WithMS is TRUE, will append '.sss' for milliseconds resolution

procedure DateTimeToIso8601StringVar(DT: TDateTime; FirstChar: AnsiChar; var result: string; WithMS: boolean=false);
Write a TDateTime into strict ISO-8601 date and/or time text
- if DT=0, returns ''
- if DT contains only a date, returns the date encoded as 'YYYY-MM-DD'
- if DT contains only a time, returns the time encoded as 'Thh:mm:ss'
- otherwise, returns the ISO-8601 date and time encoded as 'YYYY-MM-DDThh:mm:ss'
- if WithMS is TRUE, will append '.sss' for milliseconds resolution
- used e.g. by TPropInfo.GetValue() and TPropInfo.NormalizeValue() methods
function DateTimeToIso8601Text(DT: TDateTime; FirstChar: AnsiChar='T'; WithMS: boolean=false): RawUTF8;

Write a TDateTime into strict ISO-8601 date and/or time text
- if DT=0, returns "
- if DT contains only a date, returns the date encoded as 'YYYY-MM-DD'
- if DT contains only a time, returns the time encoded as 'Thh:mm:ss'
- otherwise, returns the ISO-8601 date and time encoded as 'YYYY-MM-DDThh:mm:ss'
- if WithMS is TRUE, will append '.sss' for milliseconds resolution
- used e.g. by TPropInfo.GetValue() and TPropInfo.NormalizeValue() methods

procedure DateTimeToIso8601TextVar(DT: TDateTime; FirstChar: AnsiChar; var result: RawUTF8; WithMS: boolean=false);

Write a TDateTime into strict ISO-8601 date and/or time text
- if DT=0, returns "
- if DT contains only a date, returns the date encoded as 'YYYY-MM-DD'
- if DT contains only a time, returns the time encoded as 'Thh:mm:ss'
- otherwise, returns the ISO-8601 date and time encoded as 'YYYY-MM-DDThh:mm:ss'
- if WithMS is TRUE, will append '.sss' for milliseconds resolution
- used e.g. by TPropInfo.GetValue() and TPropInfo.NormalizeValue() methods

function DateTimeToUnixMSTime(const AValue: TDateTime): TUnixMSTime;

Convert a TDateTime into a millisecond-based c-encoded time (from Unix epoch 1/1/1970)
- if AValue is 0, will return 0 (since is likely to be an error constant)

function DateTimeToUnixTime(const AValue: TDateTime): TUnixTime;

Convert a TDateTime into a second-based c-encoded time
- i.e. TDateTime into number of seconds elapsed since Unix epoch 1/1/1970

function DateToIso8601(Date: TDateTime; Expanded: boolean): RawUTF8; overload;

Basic Date conversion into ISO-8601
- use 'YYYYMMDD' format if not Expanded
- use 'YYYY-MM-DD' format if Expanded

function DateToIso8601(Y,M,D: cardinal; Expanded: boolean): RawUTF8; overload;

Basic Date conversion into ISO-8601
- use 'YYYYMMDD' format if not Expanded
- use 'YYYY-MM-DD' format if Expanded

function DateToIso8601PChar(P: PUTF8Char; Expanded: boolean; Y,M,D: PtrUInt): PUTF8Char; overload;

Write a Date to P^ Ansi buffer
- if Expanded is false, 'YYYYMMDD' date format is used
- if Expanded is true, 'YYYY-MM-DD' date format is used

function DateToIso8601PChar(Date: TDateTime; P: PUTF8Char; Expanded: boolean): PUTF8Char; overload;

Write a Date/Time to P^ Ansi buffer

function DateToIso8601Text(Date: TDateTime): RawUTF8;

Convert a date into 'YYYY-MM-DD' date format
- resulting text is compatible with all ISO-8601 functions
function DaysToIso8601(Days: cardinal; Expanded: boolean): RawUTF8;

Basic Date period conversion into ISO-8601
- will convert an elapsed number of days as ISO-8601 text
- use 'YYYYMMDD' format if not Expanded
- use 'YYYY-MM-DD' format if Expanded

function DeduplicateInt64(var Values: TInt64DynArray; Count: integer): integer;

Sort and remove any 64-bit duplicated integer from Values[]
- returns the new Values[] length

procedure DeduplicateInt64(var Values: TInt64DynArray); overload;

Sort and remove any 64-bit duplicated integer from Values[]

function DeduplicateInt64Sorted(val: PInt64Array; last: PtrInt): PtrInt;

Low-level function called by DeduplicateInt64()
- warning: caller should ensure that last>0

procedure DeduplicateInteger(var Values: TIntegerDynArray); overload;

Sort and remove any 32-bit duplicated integer from Values[]

function DeduplicateIntegerSorted(val: PIntegerArray; last: PtrInt): PtrInt;

Low-level function called by DeduplicateInteger()

procedure DeleteCriticalSectionIfNeeded(var CS: TRTLCriticalSection);

On need finalization of a mutex
- if the supplied mutex has been initialized, delete it
- if the supplied mutex is void (i.e. all filled with 0), do nothing

procedure DeleteInt64(var Values: TInt64DynArray; var ValuesCount: Integer; Index: PtrInt); overload;

Delete any 64-bit integer in Values[]

procedure DeleteInt64(var Values: TInt64DynArray; Index: PtrInt); overload;

Delete any 64-bit integer in Values[]

procedure DeleteInteger(var Values: TIntegerDynArray; Index: PtrInt); overload;

Delete any 32-bit integer in Values[]

procedure DeleteInteger(var Values: TIntegerDynArray; var ValuesCount: Integer; Index: PtrInt); overload;

Delete any 32-bit integer in Values[]

function DeleteRawUTF8(var Values: TRawUTF8DynArray; Index: integer): boolean;

Delete a RawUTF8 item in a dynamic array of RawUTF8;
function DeleteRawUTF8(var Values: TRawUTF8DynArray; var ValuesCount: integer; Index: integer; CoValues: PIntegerDynArray=nil): boolean; overload;

Delete a RawUTF8 item in a dynamic array of RawUTF8
- if CoValues is set, the integer item at the same index is also deleted

function DeleteSection(SectionFirstLine: PUTF8Char; var Content: RawUTF8; EraseSectionHeader: boolean=true): boolean; overload;

Delete a whole [Section]
- if EraseSectionHeader is TRUE (default), then the [Section] line is also deleted together with its content lines
- return TRUE if something was changed in Content
- return FALSE if [Section] doesn't exist or is already void
- SectionFirstLine may have been obtained by FindSectionFirstLine() function above

function DeleteSection(var Content: RawUTF8; const SectionName: RawUTF8; EraseSectionHeader: boolean=true): boolean; overload;

Delete a whole [Section]
- if EraseSectionHeader is TRUE (default), then the [Section] line is also deleted together with its content lines
- return TRUE if something was changed in Content
- return FALSE if [Section] doesn't exist or is already void

procedure DeleteWord(var Values: TWordDynArray; Index: PInt);

Delete any 16-bit integer in Values[]

function DirectoryDelete(const Directory: TFileName; const Mask: TFileName=FILES_ALL; DeleteOnlyFilesNotDirectory: Boolean=false; DeletedCount: PInteger=nil): Boolean;

Delete the content of a specified directory
- only one level of file is deleted within the folder: no recursive deletion is processed by this function (for safety)
- if DeleteOnlyFilesNotDirectory is TRUE, it won't remove the folder itself, but just the files found in it

function DirectoryDeleteOlderFiles(const Directory: TFileName; TimePeriod: TDateTime; const Mask: TFileName=FILES_ALL; Recursive: Boolean=false; TotalSize: PInt64=nil): Boolean;

Delete the files older than a given age in a specified directory
- for instance, to delete all files older than one day:
  DirectoryDeleteOlderFiles(FolderName, 1);
- only one level of file is deleted within the folder: no recursive deletion is processed by this function, unless Recursive is TRUE
- if Recursive=true, caller should set TotalSize^=0 to have an accurate value

procedure Div100(Y: cardinal; var res: TDiv100Rec);

Simple wrapper to efficiently compute both division and modulo per 100
- compute result.D = Y div 100 and result.M = Y mod 100
- under FPC, will use fast multiplication by reciprocal so can be inlined
- under Delphi, we use our own optimized asm version (which can't be inlined)
**function** DocVariantData(const DocVariant: variant): PDocVariantData;

Direct access to a TDocVariantData from a given variant instance
- return a pointer to the TDocVariantData corresponding to the variant instance, which may be of kind varByRef (e.g. when retrieved by late binding)
- raise an EDocVariant exception if the instance is not a TDocVariant
- the following direct trans-typing may fail, e.g. for varByRef value:
  TDocVariantData(aVarDoc.ArrayProp).Add('new item');
- so you can write the following:
  DocVariantData(aVarDoc.ArrayProp).AddItem('new item');

**function** DoubleToJSON(var tmp: ShortString; Value: double; NoExp: boolean): PShortString;

Convert a 64-bit floating-point value to its JSON text equivalency
- on Delphi Win32, calls FloatToText() in ffGeneral mode
- on other platforms, i.e. Delphi Win64 and all FPC targets, will use our own faster Fabian Loitsch's Grisu algorithm
- returns the number as text (stored into tmp variable), or "Infinity", "-Infinity", and "NaN" for corresponding IEEE special values
- result is a PShortString either over tmp, or JSON_NAN[]

**function** DoubleToShort(var S: ShortString; const Value: double): integer;

Convert a 64-bit floating-point value to its numerical text equivalency
- on Delphi Win32, calls FloatToText() in ffGeneral mode
- on other platforms, i.e. Delphi Win64 and all FPC targets, will use our own faster Fabian Loitsch's Grisu algorithm implementation
- returns the count of chars stored into S, i.e. length(S)

**function** DoubleToShortNoExp(var S: ShortString; const Value: double): integer;

Convert a 64-bit floating-point value to its numerical text equivalency without scientific notation
- on Delphi Win32, calls FloatToText() in ffGeneral mode
- on other platforms, i.e. Delphi Win64 and all FPC targets, will use our own faster Fabian Loitsch's Grisu algorithm implementation
- returns the count of chars stored into S, i.e. length(S)

**procedure** DoubleToStr(Value: Double; var result: RawUTF8); overload;

Convert a 64-bit floating-point value to its numerical text equivalency

**function** DoubleToStr(Value: Double): RawUTF8; overload;

Convert a 64-bit floating-point value to its numerical text equivalency

**function** DoubleToString(Value: Double): string;

Convert a floating-point value to its numerical text equivalency
**function** DynArray(aTypeInfo: pointer; var aValue; aCountPointer: PInteger=nil): TDynArray;

Initialize the structure with a one-dimension dynamic array
- the dynamic array must have been defined with its own type (e.g. TIntegerDynArray = array of Integer)
- if aCountPointer is set, it will be used instead of length() to store the dynamic array items count - it will be much faster when adding elements to the array, because the dynamic array won’t need to be resized each time - but in this case, you should use the Count property instead of length(array) or high(array) when accessing the data: in fact length(array) will store the memory size reserved, not the items count
- if aCountPointer is set, its content will be set to 0, whatever the array length is, or the current aCountPointer^ value is
- a typical usage could be:

```pascal
var IntArray: TIntegerDynArray;
begin
  with DynArray(TypeInfo(TIntegerDynArray), IntArray) do
  begin
    (...)
    end;
end;
```

**function** DynArrayBlobSaveJSON(TypeInfo, BlobValue: pointer): RawUTF8;

Serialize a dynamic array content, supplied as raw binary buffer, as JSON
- Value shall be set to the source dynamic array field
- is just a wrapper around TTextWriter.AddDynArrayJSON(), creating a temporary TDynArray wrapper on the stack
- to be used e.g. for custom record JSON serialization, within a TDynArrayJSONCustomWriter callback or RegisterCustomJSONSerializerFromText()

**procedure** DynArrayCopy(var Dest; const Source; SourceMaxElem: integer; TypeInfo: pointer);

Copy a dynamic array content from source to Dest
- uses internally the TDynArray.CopyFrom() method and two temporary TDynArray wrappers

**function** DynArrayElementTypeTypeName(TypeInfo: pointer; ElemTypeInfo: PPointer=nil; ExactType: boolean=false): RawUTF8;

Compute a dynamic array element information
- will raise an exception if the supplied RTTI is not a dynamic array
- will return the element type name and set ElemTypeInfo otherwise
- if there is no element type information, an approximative element type name will be returned (e.g. 'byte' for an array of 1 byte items), and ElemTypeInfo will be set to nil
- this low-level function is used e.g. by mORMotWrappers unit

**function** DynArrayEquals(TypeInfo: pointer; var Array1, Array2; Array1Count: PInteger=nil; Array2Count: PInteger=nil): boolean;

Compare two dynamic arrays by calling TDynArray.Equals

**function** DynArrayItemTypeIsSimpleBinary(const aDynArrayTypeName: RawUTF8): boolean;

Was dynamic array item after RegisterCustomJSONSerializerFromTextBinaryType() - calls DynArrayItemTypeLen() to guess the internal type name
function DynArrayItemTypeLen(const aDynArrayTypeName: RawUTF8): PtrInt;
Trim ending 'DynArray' or 's' chars from a dynamic array type name
- used internally to guess the associated item type name

function DynArrayLoad(var Value; Source: PAnsiChar; TypeInfo: pointer): PAnsiChar;
Fill a dynamic array content from a binary serialization as saved by DynArraySave() / TDynArray.Save()
- Value shall be set to the target dynamic array field
- just a function helper around TDynArray.Init + TDynArray.*

function DynArrayLoadJSON(var Value; const JSON: RawUTF8; TypeInfo: pointer): boolean; overload;
Fill a dynamic array content from a JSON serialization as saved by TTextWriter.AddDynArrayJSON, which won't be modified
- this overloaded function will make a private copy before parsing it, so is safe with a read/only or shared string - but slightly slower

function DynArrayLoadJSONArray(var Value; JSON: PUTF8Char; TypeInfo: pointer; EndOfObject: PUTF8Char=nil): PUTF8Char; overload;
Fill a dynamic array content from a JSON serialization as saved by TTextWriter.AddDynArrayJSON
- Value shall be set to the target dynamic array field
- is just a wrapper around TDynArray.LoadFromJSON(), creating a temporary TDynArray wrapper on the stack
- return a pointer at the end of the data read from JSON, nil in case of an invalid input buffer
- to be used e.g. for custom record JSON unserialization, within a TDynArrayJSONArrayCustomReader callback
- warning: the JSON buffer will be modified in-place during process - use a temporary copy if you need to access it later or if the string comes from a constant (refcount=-1) - see e.g. the overloaded DynArrayLoadJSON()

function DynArraySave(var Value; TypeInfo: pointer): RawByteString;
Serialize a dynamic array content as binary, ready to be loaded by DynArrayLoad() / TDynArray.Load()
- Value shall be set to the source dynamic array field
- just a function helper around TDynArray.Init + TDynArray.SaveTo

function DynArraySaveJSON(const Value; TypeInfo: pointer; EnumSetsAsText: boolean=false): RawUTF8;
Serialize a dynamic array content as JSON
- Value shall be set to the source dynamic array field
- is just a wrapper around TTextWriter.AddDynArrayJSON(), creating a temporary TDynArray wrapper on the stack
- to be used e.g. for custom record JSON serialization, within a TDynArrayJSONArrayCustomWriter callback or RegisterCustomJSONSerializerFromText() (following EnumSetsAsText optional parameter for nested enumerates and sets)

procedure DynArraySortIndexed(Values: pointer; ElemSize, Count: Integer; out Indexes: TSynTempBuffer; Compare: TDynArraySortCompare);
Sort any dynamic array, via an external array of indexes
- this function will use the supplied TSynTempBuffer for index storage, so use PIntegerArray(Indexes.buf) to access the values
- caller should always make Indexes.Done once done
function DynArrayTypeInfoToRecordInfo(aDynArrayTypeInfo: pointer; aDataSize: PInteger=nil): pointer;

*Retrieve the item type information of a dynamic array low-level RTTI*

function Elapsed(var PreviousTix: Int64; Interval: Integer): Boolean;

*Check if the current timestamp, in ms, matched a given period*
- will compare the current GetTickCount64 to the supplied PreviousTix
- returns TRUE if the Internal ms period was not elapsed
- returns TRUE, and set PreviousTix, if the Interval ms period was elapsed
- possible use case may be:

```pascal
var Last: Int64;
...
Last := GetTickCount64;
repeat
  ...
  if Elapsed(Last,1000) then begin
    ... // do something every second
  end;
  until Terminated;
...
```

function EndWith(const text, upText: RawUTF8): boolean;

*Check matching ending of p^ in upText*
- returns true if the item matched
- ignore case - upText^ must be already Upper
- chars are compared as 7 bit Ansi only (no accentuated characters)

function EndWithArray(const text: RawUTF8; const upArray: array of RawUTF8): integer;

*Returns the index of a matching ending of p^ in upArray[]*
- returns -1 if no item matched
- ignore case - upArray^ must be already Upper
- chars are compared as 7 bit Ansi only (no accentuated characters)

function EnsureDirectoryExists(const Directory: TFileName; RaiseExceptionOnCreationFailure: boolean=false): TFileName;

*Creates a directory if not already existing*
- returns the full expanded directory name, including trailing backslash
- returns '' on error, unless RaiseExceptionOnCreationFailure=true

function EscapeBuffer(s,d: PAnsiChar; len,max: integer): PAnsiChar;

*Fast conversion from binary data to escaped text*
- non printable characters will be written as $xx hexadecimal codes
- will be #0 terminated, with '.' characters trailing on overflow
- ensure the destination buffer contains at least max*3+3 bytes, which is always the case when using LogEscape() and its local TLogEscape variable

function EscapeToShort(const source: RawByteString): shortstring; overload;

*Fill a shortstring with the (hexadecimal) chars of the input text/binary*

function EscapeToShort(source: PAnsiChar; sourcelen: integer): shortstring; overload;

*Fill a shortstring with the (hexadecimal) chars of the input text/binary*
function EventEquals(const eventA, eventB): boolean;

Compare two TMethod instances

procedure ExcludeInt64(var Values, Excluded: TInt64DynArray; ExcludedSortSize: Integer=32);

Remove some 64-bit integer from Values[]
- Excluded is declared as var, since it will be sorted in-place during process if it contains more than ExcludedSortSize items (i.e. if the sort is worth it)

procedure ExcludeInteger(var Values, Excluded: TIntegerDynArray; ExcludedSortSize: Integer=32);

Remove some 32-bit integer from Values[]
- Excluded is declared as var, since it will be sorted in-place during process if it contains more than ExcludedSortSize items (i.e. if the sort is worth it)

function ExistsIniName(P: PUTF8Char; UpperName: PAnsiChar): boolean;

Return TRUE if Value of UpperName does exist in P, till end of current section
- expect UpperName as 'NAME='

function ExistsIniNameValue(P: PUTF8Char; const UpperName: RawUTF8; const UpperValues: array of PAnsiChar): boolean;

Return TRUE if one of the Value of UpperName exists in P, till end of current section
- expect UpperName as 'CONTENT-TYPE: '
- expect UpperValues to be any upper value with left side matching, e.g. as used by IsHTMLContentTypeTextual() function:
  result := ExistsIniNameValue(htmlHeaders,HEADER_CONTENT_TYPE_UPPER,
                  ['TEXT/','APPLICATION/JSON','APPLICATION/XML']);
- warning: this function calls IdemPCharArray(), so expects UpperValues[] items to have AT LEAST TWO CHARS (it will use fast initial 2 bytes compare)

function ExtendedToJSON(var tmp: ShortString; Value: TSynExtended; Precision: integer; NoExp: boolean): PShortString;

Convert a floating-point value to its JSON text equivalency
- depending on the platform, it may either call str() or FloatToText() in ffGeneral mode (the shortest possible decimal string using fixed or scientific format)
- returns the number as text (stored into tmp variable), or "Infinity", "-Infinity", and "NaN" for corresponding IEEE special values
- result is a PShortString either over tmp, or JSON_NAN[]

function ExtendedToShort(var S: ShortString; Value: TSynExtended; Precision: integer): integer;

Convert a floating-point value to its numerical text equivalency
- on Delphi Win32, calls FloatToText() in ffGeneral mode; on FPC uses str()
- DOUBLE_PRECISION will redirect to DoubleToShort() and its faster Fabian Loitsch's Grisu algorithm if available
- returns the count of chars stored into S, i.e. length(S)

function ExtendedToShortNoExp(var S: ShortString; Value: TSynExtended; Precision: integer): integer;

Convert a floating-point value to its numerical text equivalency without scientification notation
- DOUBLE_PRECISION will redirect to DoubleToShortNoExp() and its faster Fabian Loitsch's Grisu algorithm if available - or calls str(Value:0:precision,S)
- returns the count of chars stored into S, i.e. length(S)
procedure ExtendedToStr(Value: TSynExtended; Precision: integer; var result: RawUTF8); overload;

Convert a floating-point value to its numerical text equivalency

function ExtendedToStr(Value: TSynExtended; Precision: integer): RawUTF8; overload;

Convert a floating-point value to its numerical text equivalency

procedure FastDynArrayClear(Value: PPointer; ElemTypeInfo: pointer);

Low-level finalization of a dynamic array of variants
- faster than RTL Finalize() or setting nil

function FastFindIndexedPUTF8Char(P: PPUTF8CharArray; R: PtrInt; var SortedIndexes: TCardinalDynArray; Value: PUTF8Char; ItemComp: TUTF8Compare): PtrInt;

Retrieve the index of a PUTF8Char in a PUTF8Char array via a sort indexed
- will use fast O(log(n)) binary search algorithm

function FastFindInt64Sorted(P: PInt64Array; R: PtrInt; const Value: Int64): PtrInt;

Fast O(log(n)) binary search of a 64-bit signed integer value in a sorted array
- R is the last index of available integer entries in P^ (i.e. Count-1)
- return index of P^[result]=Value
- return -1 if Value was not found

function FastFindIntegerSorted(const Values: TIntegerDynArray; Value: integer): PtrInt; overload;

Fast O(log(n)) binary search of an integer value in a sorted integer array
- return index of Values[result]=Value
- return -1 if Value was not found

function FastFindIntegerSorted(P: PIntegerArray; R: PtrInt; Value: integer); overload;

Fast O(log(n)) binary search of an integer value in a sorted integer array
- R is the last index of available integer entries in P^ (i.e. Count-1)
- return index of P^[result]=Value
- return -1 if Value was not found

function FastFindPointerSorted(P: PPointerArray; R: PtrInt; Value: Pointer): PtrInt; overload;

Fast O(log(n)) binary search of a Pointer value in a sorted array

function FastFindPtrIntSorted(P: PPIntArray; R: PtrInt; Value: PtrInt): PtrInt; overload;

Fast O(log(n)) binary search of a PtrInt value in a sorted array

function FastFindPUTF8CharSorted(P: PPUTF8CharArray; R: PtrInt; Value: PUTF8Char): PtrInt; overload;

Retrieve the index where is located a PUTF8Char in a sorted PUTF8Char array
- R is the last index of available entries in P^ (i.e. Count-1)
- string comparison is case-sensitive StrComp (so will work with any PAnsiChar)
- returns -1 if the specified Value was not found
- will use inlined binary search algorithm with optimized x86_64 branchless asm
- slightly faster than plain FastFindPUTF8CharSorted(P,R,Value,@StrComp)
function FastFindPUTF8CharSorted(P: PPUTF8CharArray; R: PtrInt; Value: PUTF8Char; Compare: TUTF8Compare): PtrInt; overload;

    // Retrieve the index where is located a PUTF8Char in a sorted PUTF8Char array
    // - R is the last index of available entries in P^ (i.e. Count-1)
    // - string comparison will use the specified Compare function
    // - returns -1 if the specified Value was not found
    // - will use fast O(log(n)) binary search algorithm

function FastFindQWordSorted(P: PQWordArray; R: PtrInt; const Value: QWord): PtrInt;

    // Fast O(log(n)) binary search of a 64-bit unsigned integer value in a sorted array
    // - R is the last index of available integer entries in P^ (i.e. Count-1)
    // - return index of P^[result]=Value
    // - return -1 if Value was not found
    // - QWord comparison are implemented correctly under FPC or Delphi 2009+ - older compilers will
    // fast and exact SortDynArrayQWord()

function FastFindUpperPUTF8CharSorted(P: PPUTF8CharArray; R: PtrInt; Value: PUTF8Char; ValueLen: PtrInt): PtrInt;

    // Retrieve the index where is located a PUTF8Char in a sorted uppercase PUTF8Char array
    // - P[] array is expected to be already uppercased
    // - searched Value is converted to uppercase before search via UpperCopy255Buf(), so is expected
    // to be short, i.e. length < 250
    // - R is the last index of available entries in P^ (i.e. Count-1)
    // - returns -1 if the specified Value was not found
    // - will use fast O(log(n)) binary search algorithm
    // - slightly faster than plain FastFindPUTF8CharSorted(P,R,Value,@StrIComp)

function FastFindWordSorted(P: PWordArray; R: PtrInt; Value: Word): PtrInt;

    // Fast O(log(n)) binary search of a 16 bit unsigned integer value in a sorted array

function FastLocateIntegerSorted(P: PIntegerArray; R: PtrInt; Value: integer): PtrInt;

    // Retrieve the index where to insert an integer value in a sorted integer array
    // - R is the last index of available integer entries in P^ (i.e. Count-1)
    // - returns -1 if the specified Value was found (i.e. adding will duplicate a value)

function FastLocatePUTF8CharSorted(P: PPUTF8CharArray; R: PtrInt; Value: PUTF8Char; Compare: TUTF8Compare): PtrInt; overload;

    // Retrieve the index where to insert a PUTF8Char in a sorted PUTF8Char array
    // - this overloaded function accept a custom comparison function for sorting
    // - R is the last index of available entries in P^ (i.e. Count-1)
    // - string comparison is case-sensitive (so will work with any PAnsiChar)
    // - returns -1 if the specified Value was found (i.e. adding will duplicate a value)
    // - will use fast O(log(n)) binary search algorithm
function FastLocatePUTF8CharSorted(P: PPUTF8CharArray; R: PtrInt; Value: PUTF8Char): PtrInt; overload;
   Retrieve the index where to insert a PUTF8Char in a sorted PUTF8Char array
   - R is the last index of available entries in P^ (i.e. Count-1)
   - string comparison is case-sensitive StrComp (so will work with any PAnsiChar)
   - returns -1 if the specified Value was found (i.e. adding will duplicate a value)
   - will use fast O(log(n)) binary search algorithm

function FastLocateWordSorted(P: PWordArray; R: integer; Value: word): PtrInt;
   Retrieve the index where to insert a word value in a sorted word array
   - R is the last index of available integer entries in P^ (i.e. Count-1)
   - returns -1 if the specified Value was found (i.e. adding will duplicate a value)

procedure FastSetString(var s: RawUTF8; p: pointer; len: PtrInt);
   Equivalence to SetString(s,nil,len) function
   - faster especially under FPC

procedure FastSetStringCP(var s: RawUTF8; p: pointer; len, codepage: PtrInt);
   Equivalence to SetString(s,nil,len) function with a specific code page
   - faster especially under FPC

function FileAgeToDateTime(const FileName: TFileName): TDateTime;
   Get a file date and time, from its name
   - returns 0 if file doesn't exist
   - under Windows, will use GetFileAttributesEx fast API

function FileFromString(const Content: RawByteString; const FileName: TFileName; FlushOnDisk: boolean=false; FileDate: TDateTime=0): boolean;
   Create a File from a string content
   - uses RawByteString for byte storage, whatever the codepage is

function FileInfoByHandle(aFileHandle: THandle; out FileId, FileSize, LastWriteAccess, FileCreateDateDateTime: Int64): Boolean;
   Get low-level file information, in a cross-platform way
   - returns true on success
   - here file write/creation time are given as TUnixMSTime values, for better cross-platform process
   - note that FileCreateDateDateTime may not be supported by most Linux file systems, so the oldest timestamp available is returned as failover on such systems (probably the latest file metadata writing)

function FileIsSynLZ(const Name: TFileName; Magic: Cardinal): boolean;
   Returns TRUE if the supplied file name is a SynLZ compressed file, matching the Magic number as supplied to FileSynLZ() function
function FileOpenSequentialRead(const FileName: string): Integer;

Overloaded function optimized for one pass file reading
- will use e.g. the FILE_FLAG_SEQUENTIAL_SCAN flag under Windows, as stated by
  http://blogs.msdn.com/b/oldnewthing/archive/2012/01/20/10258690.aspx
- note: under XP, we observed ERROR_NO_SYSTEM_RESOURCES problems when calling FileRead()
  for chunks bigger than 32MB on files opened with this flag, so it would use regular FileOpen() on
  this deprecated OS
- under POSIX, calls plain fpOpen(FileName,O_RDONLY) which would avoid a syscall to fpFlock()
  which is not needed here
- is used e.g. by StringFromFile() and TSynMemoryStreamMapped.Create()

function FileSeek64(Handle: THandle; const Offset: Int64; Origin: cardinal): Int64;

FileSeek() overloaded function, working with huge files
- Delphi FileSeek() is buggy -> use this function to safe access files > 2 GB (thanks to sanyin for the
  report)

function FileSetDateFrom(const Dest: TFileName; SourceHandle: integer): boolean;

Copy the date of one file to another

function FileSize(const FileName: TFileName): Int64; overload;

Get a file size, from its name
- returns 0 if file doesn't exist
- under Windows, will use GetFileAttributesEx fast API

function FileSize(F: THandle): Int64; overload;

Get a file size, from its handle
- returns 0 if file doesn't exist

function FileStreamSequentialRead(const FileName: string): THandleStream;

Returns a TFileStream optimized for one pass file reading
- will use FileOpenSequentialRead(), i.e. FILE_FLAG_SEQUENTIAL_SCAN under Windows, and plain
  fpOpen(FileName, O_RDONLY) on POSIX

function FileSynLZ(const Source, Dest: TFileName; Magic: Cardinal): boolean;

Compress a file content using the SynLZ algorithm
- source file is split into 128 MB blocks for fast in-memory compression of any file size, then SynLZ
  compressed and including a Hash32 checksum
- it is not compatible with StreamSynLZ format, which has no 128 MB chunking
- you should specify a Magic number to be used to identify the compressed file format

procedure FileTimeToInt64(const FT: TFileTime; out I64: Int64);

Low-level wrapper to get the 64-bit value from a TFileTime
- as recommended by MSDN to avoid dword alignment issue

function FileTimeToUnixMSTime(const FT: TFileTime): TUnixMSTime;

Low-level conversion of a Windows 64-bit TFileTime into a Unix time ms stamp

function FileTimeToUnixTime(const FT: TFileTime): TUnixTime;

Low-level conversion of a Windows 64-bit TFileTime into a Unix time seconds stamp

function FileUnSynLZ(const Source, Dest: TFileName; Magic: Cardinal): boolean;

Uncompress a file previously compressed via FileSynLZ()
- you should specify a Magic number to be used to identify the compressed file format
procedure FillIncreasing(Values: PIntegerArray; StartValue: integer; Count: PTrUInt);

*Fill some values with i, i+1, i+2...i+Count-1*

procedure FillRandom(Dest: PCardinalArray; CardinalCount: integer; ForceGsl: boolean=false);

*Fill some memory buffer with random values*
- the destination buffer is expected to be allocated as 32-bit items
- use internally crc32c() with some rough entropy source, and Random32 gsl_rng_taus2 generator or hardware RDRAND Intel x86/x64 opcode if available (and ForceGsl is kept to its default false)
- consider using instead the cryptographic secure TAESPRNG.Main.FillRandom() method from the SynCrypto unit, or set ForceGsl=true - in particular, RDRAND is reported as very slow: see https://en.wikipedia.org/wiki/RdRand#Performance

procedure FillZero(out dig: THash256); overload;

*Fill all 32 bytes of this 256-bit buffer with zero*
- may be used to cleanup stack-allocated content
  ... finally FillZero(digest); end;

procedure FillZero(out dig: THash384); overload;

*Fill all 32 bytes of this 384-bit buffer with zero*
- may be used to cleanup stack-allocated content
  ... finally FillZero(digest); end;

procedure FillZero(var Values: TInt64DynArray); overload;

*Fill all entries of a supplied array of 64-bit integers with 0*

procedure FillZero(var dest; count: PtrInt); overload;

*Fill all bytes of a memory buffer with zero*
- is expected to be used with a constant count from SizeOf() so that inlining make it more efficient than FillCharFast(...,.,0):
  FillZero(variable,SizeOf(variable));

procedure FillZero(var secret: RawByteString); overload;

*Fill all bytes of this memory buffer with zeros, i.e. 'toto' -> #0#0#0#0#
- will write the memory buffer directly, so if this string instance is shared (i.e. has refcount>1), all other variables will contains zeros
- may be used to cleanup stack-allocated content
  ... finally FillZero(secret); end;

procedure FillZero(var secret: RawUTF8); overload;

*Fill all bytes of this UTF-8 string with zeros, i.e. 'toto' -> #0#0#0#0#
- will write the memory buffer directly, so if this string instance is shared (i.e. has refcount>1), all other variables will contains zeros
- may be used to cleanup stack-allocated content
  ... finally FillZero(secret); end;
procedure FillZero(out dig: THash512); overload;

    Fill all 64 bytes of this 512-bit buffer with zero
    - may be used to cleanup stack-allocated content
    ... finally FillZero(digest); end;

procedure FillZero(var Values: TIntegerDynArray); overload;

    Fill all entries of a supplied array of 32-bit integers with 0

procedure FillZero(var Values: TRawUTF8DynArray); overload;

    Fill all entries of a supplied array of RawUTF8 with "

procedure FillZero(var result: TGUID); overload;

    Fill a GUID with 0

procedure FillZero(var value: variant); overload;

    Fill all bytes of the value's memory buffer with zeros, i.e. 'toto' -> #0#0#0#0
    - may be used to cleanup stack-allocated content

procedure FillZero(out dig: THash128); overload;

    Fill all 16 bytes of this 128-bit buffer with zero
    - may be used to cleanup stack-allocated content
    ... finally FillZero(digest); end;

procedure FillZero(out dig: THash160); overload;

    Fill all 20 bytes of this 160-bit buffer with zero
    - may be used to cleanup stack-allocated content
    ... finally FillZero(digest); end;

function FindAnsi(A, UpperValue: PAnsiChar): boolean;

    Return true if UpperValue (Ansi) is contained in A^ (Ansi)
    - find UpperValue starting at word beginning, not inside words

function FindCSVIndex(CSV: PUTF8Char; const Value: RawUTF8; Sep: AnsiChar = ','; CaseSensitive: boolean=true; TrimValue: boolean=false): integer;

    Return the index of a Value in a CSV string
    - start at Index=0 for first one
    - return -1 if specified Value was not found in CSV items

function FindFiles(const Directory,Mask: TFileName; const IgnoreFileName: TFileName=''; SortByName: boolean=false; IncludesDir: boolean=true; SubFolder: Boolean=false): TFindFilesDynArray;

    Search for matching file names
    - just a wrapper around FindFirst/FindNext
    - you may specify several masks in Mask, e.g. as '*.jpg;*.jpeg'

function FindFilesDynArrayToFileNames(const Files: TFindFilesDynArray): TFileNameDynArray;

    Convert a result list, as returned by FindFiles(), into an array of Files[].Name
function FindIniEntry(const Content, Section, Name: RawUTF8): RawUTF8;

Find a Name= Value in a [Section] of a INI RawUTF8 Content
- this function scans the Content memory buffer, and is therefore very fast (no temporary TMemIniFile is created)
- if Section equals ",", find the Name= value before any [Section]

function FindIniEntryFile(const FileName: TFileName; const Section, Name: RawUTF8): RawUTF8;

Find a Name= Value in a [Section] of a .INI file
- if Section equals ",", find the Name= value before any [Section]
- use internally fast FindIniEntry() function above

function FindIniEntryInteger(const Content, Section, Name: RawUTF8): integer;

Find a Name= numeric Value in a [Section] of a INI RawUTF8 Content and return it as an integer, or 0 if not found
- this function scans the Content memory buffer, and is therefore very fast (no temporary TMemIniFile is created)
- if Section equals ",", find the Name= value before any [Section]

function FindIniNameValue(P: PUTF8Char; UpperName: PAnsiChar): RawUTF8;

Find the Value of UpperName in P, till end of current section
- expect UpperName as 'NAME='

function FindIniNameValueInteger(P: PUTF8Char; UpperName: PAnsiChar): PtrInt;

Find the integer Value of UpperName in P, till end of current section
- expect UpperName as 'NAME='
- return 0 if no NAME= entry was found

function FindNameValue(const NameValuePairs: RawUTF8; UpperName: PAnsiChar; var Value: RawUTF8): boolean; overload;

Search and returns a value from its uppercased named entry
- i.e. iterate IdemPChar(source,UpperName) over every line of the source
- returns true and the trimmed text just after UpperName if it has been found at line beginning
- returns false if UpperName was not found was not found at any line beginning
- could be used e.g. to efficiently extract a value from HTTP headers, whereas FindIniNameValue() is tuned for [section]-oriented INI files

function FindNameValue(P: PUTF8Char; UpperName: PAnsiChar): PUTF8Char; overload;

Search for a value from its uppercased named entry
- i.e. iterate IdemPChar(source,UpperName) over every line of the source
- returns the text just after UpperName if it has been found at line beginning
- returns nil if UpperName was not found was not found at any line beginning
- could be used as alternative to FindIniNameValue() and FindIniNameValueInteger() if there is no section, i.e. if search should not stop at '[' but at source end

function FindNextUTF8WordBegin(U: PUTF8Char): PUTF8Char;

Points to the beginning of the next word stored in U
- returns nil if reached the end of U (i.e. #0 char)
- here a "word" is a Win-Ansi word, i.e. '0'..'9', 'A'..'Z'
function FindObjectEntry(const Content, Name: RawUTF8): RawUTF8;

*Retrieve a property value in a text-encoded class*
- follows the Delphi serialized text object format, not standard .ini
- if the property is a string, the simple quotes ' are trimmed

function FindObjectEntryWithoutExt(const Content, Name: RawUTF8): RawUTF8;

*Retrieve a filename property value in a text-encoded class*
- follows the Delphi serialized text object format, not standard .ini
- if the property is a string, the simple quotes ' are trimmed
- any file path and any extension are trimmed

function FindPropName(const Names: array of RawUTF8; const Name: RawUTF8): integer;

*Return the index of Value in Values[], -1 if not found*
- here name search would use fast IdemPropNameU() function

function FindPropName(Values: PRawUTF8; const Value: RawUTF8; ValuesCount: integer): integer;

*Return the index of Value in Values[] using IdemPropNameU(), -1 if not found*
- typical use with a dynamic array is like:
  ```pascal
  index := FindPropName(pointer(aDynArray),length(aDynArray),aValue);
  ```

function FindRawUTF8(const Values: TRawUTF8DynArr; const Value: RawUTF8; CaseSensitive: boolean=true): integer;

*Return the index of Value in Values[], -1 if not found*
- CaseSensitive=false will use StrICmp() for A..Z / a..z equivalence

function FindRawUTF8(const Values: array of RawUTF8; const Value: RawUTF8; CaseSensitive: boolean=true): integer;

*Return the index of Value in Values[], -1 if not found*
- CaseSensitive=false will use StrICmp() for A..Z / a..z equivalence

function FindRawUTF8(Values: PRawUTF8; const Value: RawUTF8; ValuesCount: integer; CaseSensitive: boolean): integer;

*Low-level efficient search of Value in Values[]*
- CaseSensitive=false will use StrICmp() for A..Z / a..z equivalence

function FindSectionFirstLine(var source: PUTF8Char; search: PAnsiChar): boolean;

*Find the position of the [SEARCH] section in source*
- return true if [SEARCH] was found, and store pointer to the line after it in source

function FindSectionFirstLineW(var source: PWideChar; search: PUTF8Char): boolean;

*Find the position of the [SEARCH] section in source*
- return true if [SEARCH] was found, and store pointer to the line after it in source
- this version expects source^ to point to an Unicode char array

function FindShortStringListExact(List: PShortString; MaxValue: integer; aValue: PUTF8Char; aValueLen: PtrInt): integer;

*Fast search of an exact case-insensitive match of a RTTI's PShortString array

function FindShortStringListTrimLowerCase(List: PShortString; MaxValue: integer; aValue: PUTF8Char; aValueLen: PtrInt): integer;

*Fast case-insensitive search of a left-trimmed lowercase match of a RTTI's PShortString array
function FindShortStringListTrimLowerCaseExact(List: PShortString; MaxValue: integer; aValue: PUTF8Char; aValueLen: PtrInt): integer;

Fast case-sensitive search of a left-trimmed lowercase match of a RTTI's PShortString array

function FindUnicode(PW: PWideChar; Upper: PWideChar; UpperLen: PtrInt): boolean;

Return true if Upper (Unicode encoded) is contained in U^ (UTF-8 encoded)
- will use the slow but accurate Operating System API to perform the comparison at Unicode-level

function FindUTF8(U: PUTF8Char; UpperValue: PAnsiChar): boolean;

Return true if UpperValue (Ansi) is contained in U^ (UTF-8 encoded)
- find UpperValue starting at word beginning, not inside words
- UTF-8 decoding is done on the fly (no temporary decoding buffer is used)

function FindWinAnsiIniEntry(const Content, Section, Name: RawUTF8): RawUTF8;

Find a Name= Value in a [Section] of a INI WinAnsi Content
- same as FindIniEntry(), but the value is converted from WinAnsi into UTF-8

function FloatStrCopy(s, d: PUTF8Char): PUTF8Char;

Copy a floating-point text buffer with proper correction and validation
- will correct on the fly '.5' -> '0.5' and '-.5' -> '-0.5'
- will end not only on #0 but on any char not matching 1[.2[e-][3]] pattern
- is used when the input comes from a third-party source with no regular output, e.g. a database driver, via TTextWriter.AddFloatStr

function FloatToJSONNan(const s: ShortString): PShortString;

Recognize if the supplied text is NAN/INF/+INF/-INF, i.e. not a number
- returns the number as text (stored into tmp variable), or "Infinity", "-Infinity", and "NaN" for corresponding IEEE special values
- result is a PShortString either over tmp, or JSON_NAN[]

function FloatToShortNan(const s: shortstring): TFloatNan;

Check if the supplied text is NAN/INF/+INF/-INF, i.e. not a number
- as returned by ExtendedToShort/DoubleToShort textual conversion
- such values do appear as IEEE floating points, but are not defined in JSON

function FloatToStrNan(const s: RawUTF8): TFloatNan;

Check if the supplied text is NAN/INF/+INF/-INF, i.e. not a number
- as returned e.g. by ExtendedToStr/DoubleToStr textual conversion
- such values do appear as IEEE floating points, but are not defined in JSON

function fnv32(crc: cardinal; buf: PAnsiChar; len: PtrInt): cardinal;

Simple FNV-1a hashing function
- when run over our regression suite, is similar to crc32c() about collisions, and 4 times better than kr32(), but also slower than the others
- fnv32() is 715.5 MB/s - kr32() 898.8 MB/s
- this hash function should not be usefull, unless you need several hashing algorithms at once (e.g. if crc32c with diverse seeds is not enough)

function FormatBuffer(const Format: RawUTF8; const Args: array of const; Dest: pointer; DestLen: PtrInt): PtrInt;

Fast Format() function replacement, tuned for direct memory buffer write
- use the same single token % (and implementation) than FormatUTF8()
- returns the number of UTF-8 bytes appended to Dest^
procedure FormatShort(const Format: RawUTF8; const Args: array of const; var result: shortstring);

  *Fast Format() function replacement, for UTF-8 content stored in shortstring*
  - use the same single token % (and implementation) than FormatUTF8()
  - shortstring allows fast stack allocation, so is perfect for small content
  - truncate result if the text size exceeds 255 bytes

procedure FormatShort16(const Format: RawUTF8; const Args: array of const; var result: TShort16);

  *Fast Format() function replacement, for UTF-8 content stored in TShort16*
  - truncate result if the text size exceeds 16 bytes

function FormatString(const Format: RawUTF8; const Args: array of const): string; overload;

  *Fast Format() function replacement, tuned for small content*
  - use the same single token % (and implementation) than FormatUTF8()

procedure FormatString(const Format: RawUTF8; const Args: array of const; out result: string); overload;

  *Fast Format() function replacement, tuned for small content*
  - use the same single token % (and implementation) than FormatUTF8()

function FormatToShort(const Format: RawUTF8; const Args: array of const): shortstring;

  *Fast Format() function replacement, for UTF-8 content stored in shortstring*

function FormatUTF8(const Format: RawUTF8; const Args, Params: array of const; JSONFormat: boolean=false): RawUTF8; overload;

  *Fast Format() function replacement, handling % and ? parameters*
  - will include Args[] for every % in Format
  - will inline Params[] for every ? in Format, handling special "inlined" parameters, as exected by mORMot.pas unit, i.e. :1234: for numerical values, and :('quoted " string'): for textual values
  - if optional JSONFormat parameter is TRUE, ? parameters will be written as JSON quoted strings, without :(...): tokens, e.g. "quoted "" string"
  - resulting string has no length limit and uses fast concatenation
  - note that, due to a Delphi compiler limitation, cardinal values should be type-casted to Int64() (otherwise the integer mapped value will be converted)
  - any supplied TObject instance will be written as their class name

procedure FormatUTF8(const Format: RawUTF8; const Args: array of const; out result: RawUTF8); overload;

  *Fast Format() function replacement, optimized for RawUTF8*
  - overloaded function, which avoid a temporary RawUTF8 instance on stack
function FormatUTF8(const Format: RawUTF8; const Args: array of const): RawUTF8;

Fast Format() function replacement, optimized for RawUTF8
- only supported token is %, which will be written in the resulting string according to each Args[] supplied items - so you will never get any exception as with the SysUtils.Format() when a specifier is incorrect
- resulting string has no length limit and uses fast concatenation
- there is no escape char, so to output a '%' character, you need to use '%' as place-holder, and specify '%' as value in the Args array
- note that, due to a Delphi compiler limitation, cardinal values should be type-casted to Int64() (otherwise the integer mapped value will be converted)
- any supplied TObject instance will be written as their class name

procedure FormatUTF8ToVariant(const Fmt: RawUTF8; const Args: array of const; var Value: variant);

Convert a FormatUTF8() UTF-8 encoded string into a variant RawUTF8 varString

function FromI32(const Values: array of integer): TIntegerFieldDynArray;
Initializes a dynamic array from a set of 32-bit integer signed values

function FromI64(const Values: array of Int64): TInt64DynArray;
Initializes a dynamic array from a set of 64-bit integer signed values

function FromU32(const Values: array of cardinal): TCardinalDynArray;
Initializes a dynamic array from a set of 32-bit integer unsigned values

function FromU64(const Values: array of QWord): TQWordDynArray;
Initializes a dynamic array from a set of 64-bit integer unsigned values

function FromVarBlob(Data: PByte): TValueResult;
Retrieve pointer and length to a variable-length text/blob buffer

function FromVarInt32(var Source: PByte): integer;
Convert a 32-bit variable-length integer buffer into an integer
- decode negative values from cardinal two-complement, i.e. 0=0,1=1,2=-1,3=2,4=-2...

function FromVarInt64(var Source: PByte): Int64;
Convert a 64-bit variable-length integer buffer into a Int64

function FromVarInt64Value(Source: PByte): Int64;
Convert a 64-bit variable-length integer buffer into a Int64
- this version won't update the Source pointer

procedure FromVarString(var Source: PByte; var Value: TSynTempBuffer); overload;
Retrieve a variable-length UTF-8 encoded text buffer in a temporary buffer
- caller should call Value.Done after use of the Value.buf memory
- this overloaded function would include a trailing #0, so Value.buf could be parsed as a valid PUTF8Char buffer (e.g. containing JSON)
### function FromVarString

**Declaration:**
```
function FromVarString(var Source: PByte; SourceMax: PByte; var Value: TSynTempBuffer): boolean; overload;
```

**Description:**
- Retrieve a variable-length UTF-8 encoded text buffer in a temporary buffer
- caller should call Value.Done after use of the Value.buf memory
- this overloaded function will also check for the SourceMax end of buffer, returning TRUE on success, or FALSE on any buffer overload detection

### function FromVarString

**Declaration:**
```
function FromVarString(var Source: PByte; SourceMax: PByte; var Value: RawByteString; CodePage: integer): boolean; overload;
```

**Description:**
- Retrieve a variable-length text buffer
- this overloaded function will set the supplied code page to the AnsiString and will also check for the SourceMax end of buffer
- returns TRUE on success, or FALSE on any buffer overload detection

### function FromVarString

**Declaration:**
```
function FromVarString(var Source: PByte): RawUTF8; overload;
```

**Description:**
- Retrieve a variable-length UTF-8 encoded text buffer in a newly allocation RawUTF8

### function FromVarString

**Declaration:**
```
function FromVarString(var Source: PByte; SourceMax: PByte): RawUTF8; overload;
```

**Description:**
- Safe retrieve a variable-length UTF-8 encoded text buffer in a newly allocation RawUTF8
- supplied SourceMax value will avoid any potential buffer overflow

### procedure FromVarString

**Declaration:**
```
procedure FromVarString(var Source: PByte; var Value: RawByteString; CodePage: integer); overload;
```

**Description:**
- Retrieve a variable-length text buffer
- this overloaded function will set the supplied code page to the AnsiString

### function FromVarUInt32

**Declaration:**
```
function FromVarUInt32(var Source: PByte): cardinal; overload;
```

**Description:**
- Convert a 32-bit variable-length integer buffer into a cardinal
- fast inlined process for any number < 128
- use overloaded FromVarUInt32() or FromVarUInt32Safe() with a SourceMax pointer to avoid any potential buffer overflow

### function FromVarUInt32

**Declaration:**
```
function FromVarUInt32(var Source: PByte; SourceMax: PByte; out Value: cardinal): boolean; overload;
```

**Description:**
- Convert a 32-bit variable-length integer buffer into a cardinal
- will call FromVarUInt32() if SourceMax=nil, or FromVarUInt32Safe() if set
- returns false on error, true if Value has been set properly

### function FromVarUInt32

**Declaration:**
```
function FromVarUInt32Big(var Source: PByte): cardinal;
```

**Description:**
- Convert a 32-bit variable-length integer buffer into a cardinal
- this version could be called if number is likely to be > $7f, so it inlining the first byte won't make any benefit

### function FromVarUInt32

**Declaration:**
```
function FromVarUInt32High(var Source: PByte): cardinal;
```

**Description:**
- Convert a 32-bit variable-length integer buffer into a cardinal
- this version must be called if Source^ has already been checked to be > $7f

### function FromVarUInt32

**Declaration:**
```
function FromVarUInt32Safe(Source, SourceMax: PByte; out Value: cardinal): PByte;
```

**Description:**
- Safely convert a 32-bit variable-length integer buffer into a cardinal
- slower but safer process checking out of boundaries memory access in Source
- SourceMax is expected to be not nil, and to point to the first byte just after the Source memory buffer
- returns nil on error, or point to next input data on successful decoding
function FromVarUInt32Up128(var Source: PByte): cardinal;

Convert a 32-bit variable-length integer buffer into a cardinal
- used e.g. when inlining FromVarUInt32()
- this version must be called if Source^ has already been checked to be > $7f
result := Source^;
inc(Source);
if result>$7f then
result := (result and $7F) or FromVarUInt32Up128(Source);

function FromVarUInt64(var Source: PByte; SourceMax: PByte; out Value: Qword): boolean; overload;

Convert a 64-bit variable-length integer buffer into a UInt64
- will call FromVarUInt64() if SourceMax=nil, or FromVarUInt64Safe() if set
- returns false on error, true if Value has been set properly

function FromVarUInt64(var Source: PByte): QWord; overload;

Convert a 64-bit variable-length integer buffer into a UInt64

function FromVarUInt64Safe(Source, SourceMax: PByte; out Value: QWord): PByte;
Safely convert a 64-bit variable-length integer buffer into a UInt64
- slower but safer process checking out of boundaries memory access in Source
- SourceMax is expected to be not nil, and to point to the first byte just after the Source memory buffer
- returns nil on error, or point to next input data on successful decoding

procedure FromVarVariant(var Source: PByte; var Value: variant; CustomVariantOptions: PDocVariantOptions=nil);

Retrieve a variant value from variable-length buffer
- matches TFileBufferWriter.Write()
- how custom type variants are created can be defined via CustomVariantOptions
- is just a wrapper around VariantLoad()

procedure GarbageCollectorFree;

Force the global "Garbage collector" list to be released immediately
- this function is called in the finalization section of this unit
- you should NEVER have to call this function, unless some specific cases (e.g. when using Delphi packages, just before releasing the package)

procedure GarbageCollectorFreeAndNil(var InstanceVariable; Instance: TObject);

A global "Garbage collector" for some TObject global variables which must live during whole main executable process
- this list expects a pointer to the TObject instance variable to be specified, and will be set to nil (like a FreeAndNil)
- this may be useful when used when targetting Delphi IDE packages, to circumvent the bug of duplicated finalization of units, in the scope of global variables
- to be used, e.g. as:
if SynAnsiConvertList=nil then
  GarbageCollectorFreeAndNil(SynAnsiConvertList,TObjectList.Create);

function gcd(a, b: cardinal): cardinal;

Compute GCD of two integers using substraction-based Euclidean algorithm
function GetAllBits(Bits, BitCount: cardinal): boolean;
Returns TRUE if all BitCount bits are set in the input 32-bit cardinal

function GetBit(const Bits; aIndex: PtrInt): boolean;
Retrieve a particular bit status from a bit array
- this function can't be inlined, whereas GetBitPtr() function can

function GetBit64(const Bits: Int64; aIndex: PtrInt): boolean;
Retrieve a particular bit status from a 64-bit integer bits (max aIndex is 63)

function GetBitCSV(const Bits; BitsCount: integer): RawUTF8;
Convert a set of bit into a CSV content
- each bit is stored as BitIndex+1, and separated by a ','
- several bits set to one can be regrouped via 'first-last,' syntax
- ','0' is always appended at the end of the CSV chunk to mark its end

function GetBitPtr(Bits: pointer; aIndex: PtrInt): boolean;
Retrieve a particular bit status from a bit array
- GetBit() can't be inlined, whereas this pointer-oriented function can

function GetBitsCount(const Bits; Count: PtrInt): PtrInt;
Compute the number of bits set in a bit array
- Count is the bit count, not byte size
- will use fast SSE4.2 popcnt instruction if available on the CPU

function GetBitsCountPas(value: PtrInt): PtrInt;
Pure pascal version of GetBitsCountPtrInt()
- defined just for regression tests - call GetBitsCountPtrInt() instead
- has optimized asm on x86_64 and i386

function GetBitsCountSSE42(value: PtrInt): PtrInt;
SSE 4.2 version of GetBitsCountPtrInt()
- defined just for regression tests - call GetBitsCountPtrInt() instead

function GetBoolean(P: PUTF8Char): boolean;
Get a boolean value stored as true/false text in P^ 
- would also recognize any non 0 integer as true

function GetCaptionFromClass(C: TClass): string;
UnCamelCase and translate the class name, trimming any left 'T', 'TSyn', 'TSQL' or 'TSQLRecord'
- return generic VCL string type, i.e. UnicodeString for Delphi 2009+

function GetCaptionFromEnum(aTypeInfo: pointer; aIndex: integer): string;
UnCamelCase and translate the enumeration item

procedure GetCaptionFromPCharLen(P: PUTF8Char; out result: string);
UnCamelCase and translate a char buffer
- P is expected to be #0 ended
- return "string" type, i.e. UnicodeString for Delphi 2009+

procedure GetCaptionFromTrimmed(PS: PShortString; var result: string);
Low-level helper to retrieve a (translated) caption from a PShortString
- as used e.g. by GetEnumCaptions or GetCaptionFromEnum
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetCardinal(P: PUTF8Char): PtrUInt;</td>
<td>Get the unsigned 32-bit integer value stored in P^</td>
</tr>
<tr>
<td>GetCardinalDef(P: PUTF8Char; Default: PtrUInt): PtrUInt;</td>
<td>Get the unsigned 32-bit integer value stored in P^</td>
</tr>
<tr>
<td>GetCardinalW(P: PWideChar): PtrUInt;</td>
<td>Get the unsigned 32-bit integer value stored as Unicode string in P^</td>
</tr>
<tr>
<td>GetClassParent(C: TClass): TClass;</td>
<td>Just a wrapper around vmtParent to avoid a function call</td>
</tr>
<tr>
<td>GetCSVItem(P: PUTF8Char; Index: PtrUInt; Sep: AnsiChar=','): RawUTF8;</td>
<td>Return n-th indexed CSV string in P, starting at Index=0 for first one</td>
</tr>
<tr>
<td>GetCSVItemString(P: PChar; Index: PtrUInt; Sep: Char=','): string;</td>
<td>Return n-th indexed CSV string in P, starting at Index=0 for first one</td>
</tr>
<tr>
<td>GetDelphiCompilerVersion: RawUTF8;</td>
<td>Return the Delphi/FPC Compiler Version</td>
</tr>
<tr>
<td>GetDisplayNameFromClass(C: TClass): RawUTF8;</td>
<td>Will get a class name as UTF-8</td>
</tr>
<tr>
<td>GetEnumCaptions(aTypeInfo: pointer; aDest: PString);</td>
<td>Helper to retrieve all (translated) caption texts of an enumerate</td>
</tr>
<tr>
<td>GetEnumName(aTypeInfo: pointer; aIndex: integer): PShortString;</td>
<td>Helper to retrieve the text of an enumerate item</td>
</tr>
<tr>
<td>GetEnumNames(aTypeInfo: pointer; aDest: PPShortString);</td>
<td>Helper to retrieve all texts of an enumerate</td>
</tr>
<tr>
<td>GetEnumNameValue(aTypeInfo: pointer; const aValue: RawUTF8; AlsoTrimLowerCase: boolean=false): Integer; overload;</td>
<td>Helper to retrieve the index of an enumerate item from its text</td>
</tr>
</tbody>
</table>
function GetEnumNameValue(aTypeInfo: pointer; aValue: PUTF8Char; aValueLen: PtrInt; AlsoTrimLowerCase: boolean=false): Integer; overload;
   Helper to retrieve the index of an enumerate item from its text
   - returns -1 if aValue was not found
   - will search for the exact text and also trim the lowercase 'a'..'z' chars on left side of the text if no exact match is found and AlsoTrimLowerCase is TRUE
   - see also RTTI related classes of mORMot.pas unit, e.g. TEnumType

function GetEnumNameValueTrimmed(aTypeInfo: pointer; aValue: PUTF8Char; aValueLen: PtrInt): integer;
   Retrieve the index of an enumerate item from its left-trimmed text
   - text comparison is case-insensitive for A-Z characters
   - will trim the lowercase 'a'..'z' chars on left side of the supplied aValue text
   - returns -1 if aValue was not found

function GetEnumNameValueTrimmedExact(aTypeInfo: pointer; aValue: PUTF8Char; aValueLen: PtrInt): integer;
   Retrieve the index of an enumerate item from its left-trimmed text
   - text comparison is case-sensitive for A-Z characters
   - will trim the lowercase 'a'..'z' chars on left side of the supplied aValue text
   - returns -1 if aValue was not found

procedure GetEnumTrimmedNames(aTypeInfo: pointer; aDest: PRawUTF8); overload;
   Helper to retrieve all trimmed texts of an enumerate
   - may be used as cache to retrieve UTF-8 text without lowercase 'a'..'z' chars

function GetEnumTrimmedNames(aTypeInfo: pointer): TRawUTF8DynArray; overload;
   Helper to retrieve all trimmed texts of an enumerate as UTF-8 strings

function GetExtended(P: PUTF8Char): TSynExtended; overload;
   Get the extended floating point value stored in P^ 
   - this overloaded version returns 0 as a result if the content of P is invalid

function GetExtended(P: PUTF8Char; var err: integer): TSynExtended; overload;
   Get the extended floating point value stored in P^ 
   - set the err content to the index of any faulty character, 0 if conversion was successful (same as the standard val function)

function GetFileNameExtIndex(const FileName, CSVExt: TFileName): integer;
   Extract a file extension from a file name, then compare with a comma separated list of extensions
   - e.g. GetFileNameExtIndex('test.log','exe,log,map')=1
   - will return -1 if no file extension match
   - will return any matching extension, starting count at 0
   - extension match is case-insensitive

function GetFileNameWithoutExt(const FileName: TFileName; Extension: PFileName=nil): TFileName;
   Extract file name, without its extension
   - may optionally return the associated extension, as '.ext'
### function GetHighUTF8UCS4(var U: PUTF8Char): PtrUInt;

*Internal function, used to retrieve a UCS4 char (>127) from UTF-8*
- not to be called directly, but from inlined higher-level functions
- here U^ shall be always >= #80
- typical use is as such:
  ```haskell
  ch := ord(P^);
  if ch and $80=0 then
    inc(P) else
  ch := GetHighUTF8UCS4(P);
  ```

### function GetInt64(P: PUTF8Char; var err: integer): Int64; overload;

*Get the 64-bit signed integer value stored in P^*
- set the err content to the index of any faulty character, 0 if conversion was successful (same as the standard val function)

### function GetInt64(P: PUTF8Char): Int64; overload;

*Get the 64-bit integer value stored in P^*

### function GetInt64Def(P: PUTF8Char; const Default: Int64): Int64;

*Get the 64-bit integer value stored in P^*
- if P if nil or not start with a valid numerical value, returns Default

### function GetInteger(P: PUTF8Char; var err: integer): PtrInt; overload;

*Get the signed 32-bit integer value stored in P^*
- this version return 0 in err if no error occured, and 1 if an invalid character was found, not its exact index as for the val() function

### function GetInteger(P, PEnd: PUTF8Char): PtrInt; overload;

*Get the signed 32-bit integer value stored in P^..PEnd^*
- will end parsing when P^ does not contain any number (e.g. it reaches any ending #0 char), or when P reached PEnd (avoiding any buffer overflow)

### function GetInteger(P: PUTF8Char): PtrInt; overload;

*Get the signed 32-bit integer value stored in P^*
- we use the PtrInt result type, even if expected to be 32-bit, to use native CPU register size (don't want any 32-bit overflow here)
- will end parsing when P^ does not contain any number (e.g. it reaches any ending #0 char)

### function GetIntegerDef(P: PUTF8Char; Default: PtrInt): PtrInt;

*Get the signed 32-bit integer value stored in P^*
- if P if nil or not start with a valid numerical value, returns Default

### function GetJpegSize(const jpeg: TFileName; out Height, Width: integer): boolean; overload;

*Fast guess of the size, in pixels, of a JPEG file*
- will only scan for basic JPEG structure, up to the StartOfFrame (SOF) chunk
- returns TRUE if the buffer is likely to be a JPEG picture, and set the Height + Width variable with its dimensions - but there may be false positive recognition, and no waranty that the file is a valid JPEG picture
- returns FALSE if the file content does not have any expected SOI/SOF markers
function GetJpegSize(jpeg: PAnsiChar; len: PtrInt; out Height, Width: integer): boolean; overload;

Fast guess of the size, in pixels, of a JPEG memory buffer
- will only scan for basic JPEG structure, up to the StartOfFrame (SOF) chunk
- returns TRUE if the buffer is likely to be a JPEG picture, and set the Height + Width variable with its dimensions - but there may be false positive recognition, and no waranty that the memory buffer holds a valid JPEG picture
- returns FALSE if the buffer does not have any expected SOI/SOF markers

function GetJSONField(P: PUTF8Char; out PDest: PUTF8Char; wasString: PBoolean=nil; EndOfObject: PUTF8Char=nil; Len: PInteger=nil): PUTF8Char;

Efficient JSON field in-place decoding, within a UTF-8 encoded buffer
- this function decodes in the P^ buffer memory itself (no memory allocation or copy), for faster process - so take care that P^ is not shared
- PDest points to the next field to be decoded, or nil when end is reached
- EndOfObject (if not nil) is set to the JSON value char (',' ':' or '}') e.g.
- optional wasString is set to true if the JSON value was a JSON "string"
- "strings" are decoded as 'strings', with wasString=true, properly JSON unescaped (e.g. any \u0123 pattern would be converted into UTF-8 content)
- null is decoded as nil, with wasString=false
- true/false boolean values are returned as 'true'/false', with wasString=false
- any number value is returned as its ascii representation, with wasString=false
- works for both field names or values (e.g. "FieldName":' or 'Value,')

Used for DI-2.1.2 (page 2545).

function GetJSONFieldOrObjectOrArray(var P: PUTF8Char; wasString: PBoolean=nil; EndOfObject: PUTF8Char=nil; HandleValuesAsObjectOrArray: Boolean=false; NormalizeBoolean: Boolean=true; Len: PInteger=nil): PUTF8Char;

Decode a JSON content in an UTF-8 encoded buffer
- GetJSONField() will only handle JSON "strings" or numbers - if HandleValuesAsObjectOrArray is TRUE, this function will process JSON { objects } or [ arrays ] and add a #0 at the end of it
- this function decodes in the P^ buffer memory itself (no memory allocation or copy), for faster process - so take care that it is an unique string
- returns a pointer to the value start, and moved P to the next field to be decoded, or P=nil in case of any unexpected input
- wasString is set to true if the JSON value was a "string"
- EndOfObject (if not nil) is set to the JSON value end char (',' ':' or '}')
- if Len is set, it will contain the length of the returned pointer value

procedure GetJSONItemAsRawJSON(var P: PUTF8Char; var result: RawJSON; EndOfObject: PAnsiChar=nil);

Retrieve the next JSON item as a RawJSON variable
- buffer can be either any JSON item, i.e. a string, a number or even a JSON array (ending with ]) or a JSON object (ending with {)
- EndOfObject (if not nil) is set to the JSON value end char (',' ':' or '}')
function GetJSONItemAsRawUTF8(var P: PUTF8Char; var output: RawUTF8; wasString: PBoolean=nil; EndOfObject: PUTF8Char=nil): boolean;

Retrieve the next JSON item as a RawUTF8 decoded buffer
- buffer can be either any JSON item, i.e. a string, a number or even a JSON array (ending with "]) or a JSON object (ending with ")
- EndOfObject (if not nil) is set to the JSON value end char (',' ':' or '}
- just call GetJSONField(), and create a new RawUTF8 from the returned value, after proper unescape if wasString^=true

function GetJSONPropName(var P: PUTF8Char; Len: PInteger=nil): PUTF8Char; overload;
Decode a JSON field name in an UTF-8 encoded buffer
- this function decodes in the P^ buffer memory itself (no memory allocation or copy), for faster process - so take care that P^ is not shared
- it will return the property name (with an ending #0) or nil on error
- this function will handle strict JSON property name (i.e. a "string"), but also MongoDB extended syntax, e.g. "age:{$gt:18}" or '{people.age':{$gt:18}" see @http://docs.mongodb.org/manual/reference/mongodb-extended-json

procedure GetJSONPropName(var P: PUTF8Char; out PropName: shortstring); overload;
Decode a JSON field name in an UTF-8 encoded shortstring variable
- this function would left the P^ buffer memory untouched, so may be safer than the overloaded GetJSONPropName() function in some cases
- it will return the property name as a local UTF-8 encoded shortstring, or PropName='' on error
- this function won't unescape the property name, as strict JSON (i.e. a "string")
- but it will handle MongoDB syntax, e.g. "age:{$gt:18}" or '{people.age':{$gt:18}" see @http://docs.mongodb.org/manual/reference/mongodb-extended-json

function GetLastCSVItem(const CSV: RawUTF8; Sep: AnsiChar=','): RawUTF8;
Return last CSV string in the supplied UTF-8 content

function GetLineContains(p,pEnd, up: PUTF8Char): boolean;
Returns TRUE if the supplied uppercased text is contained in the text buffer

function GetLineSize(P,PEnd: PUTF8Char): PtrUInt;
Compute the line length from source array of chars
- if PEnd = nil, end counting at either #0, #13 or #10
- otherwise, end counting at either #13 or #10
- just a wrapper around BufferLineLength() checking PEnd=nil case

function GetLineSizeSmallerThan(P,PEnd: PUTF8Char; aMinimalCount: integer): boolean;
Returns true if the line length from source array of chars is not less than the specified count

procedure GetMemAligned(var s: RawByteString; p: pointer; len: PtrInt; out aligned: pointer);
Initialize a RawByteString, ensuring returned "aligned" pointer is 16-bytes aligned
- to be used e.g. for proper SSE process
The function `GetMimeContentType` retrieves the MIME content type from its file name or a supplied binary buffer. It performs the following steps:

1. First checks for known file extensions.
2. Inspects the binary content if no file extension is specified.
3. Returns the MIME type ready to be appended to a 'Content-Type:' HTTP header.
4. By default, it returns 'application/octet-stream' (BINARY_CONTENT_TYPE) or 'application/fileextension' if the file name was specified.
5. See the Wikipedia page on Internet Media Types for most common values.

The function `GetMimeContentTypeFromBuffer` does a similar task but looks at the first bytes of the binary buffer to guess the MIME type.

The `GetMimeContentTypeHeader` function retrieves the HTTP header for the MIME content type by appending `HEADER_CONTENT_TYPE` and the result of `GetMimeContentType`.

The `GetNextFieldProp` function retrieves the next SQL-like identifier within the UTF-8 buffer, trimming spaces and trailing ';'.

The `GetNextFieldPropSameLine` function retrieves the next identifier within the UTF-8 buffer on the same line, handling line feeds and not for regular config files with name/value pairs.

The `GetNextItem` function returns the next CSV string from the input buffer, allowing overloads for handling CSV strings with or without a separator and a quote character.

The `GetNextItemCardinal` function returns the next CSV string as an unsigned integer from the input buffer, with overloads for handling strings with or without separators and quotes.
function GetNextItemCardinalStrict(var P: PUTF8Char): PtrUInt;
    Return next CSV string as unsigned integer from P, 0 if no more
    - P^ will point to the first non digit character (the item separator, e.g. ',' for CSV)

function GetNextItemCardinalW(var P: PWideChar; Sep: WideChar=',,'): PtrUInt;
    Return next CSV string as unsigned integer from P, 0 if no more
    - this version expects P^ to point to an Unicode char array

procedure GetNextItemCurrency(var P: PUTF8Char; out result: currency; Sep: AnsiChar=',,'); overload;
    Return next CSV string as currency from P, 0.0 if no more
    - if Sep is #0, will return all characters until next whitespace char

function GetNextItemCurrency(var P: PUTF8Char; Sep: AnsiChar=',,'): currency;
    overload;
    Return next CSV string as currency from P, 0.0 if no more
    - if Sep is #0, will return all characters until next whitespace char

function GetNextItemDouble(var P: PUTF8Char; Sep: AnsiChar=',,'): double;
    Return next CSV string as double from P, 0.0 if no more
    - if Sep is #0, will return all characters until next whitespace char

function GetNextItemHexa(var P: PUTF8Char; Sep: AnsiChar=',,'): QWord;
    Return next CSV hexadecimal string as 64-bit unsigned integer from P
    - returns 0 if no valid hexadecimal text is available in P
    - if Sep is #0, it won't be searched for
    - will first fill the 64-bit value with 0, then decode each two hexadecimal characters available in P
    - could be used to decode TTextWriter.AddBinToHexDisplayMinChars() output

function GetNextItemHexDisplayToBin(var P: PUTF8Char; Bin: PByte; BinBytes: integer; Sep: AnsiChar=',,'): boolean;
    Decode next CSV hexadecimal string from P, nil if no more or not matching BinBytes
    - Bin is filled with 0 if the supplied CSV content is invalid
    - if Sep is #0, it will read the hexadecimal chars until a whitespace is reached

function GetNextItemInt64(var P: PUTF8Char; Sep: AnsiChar=',,'): Int64;
    Return next CSV string as 64-bit signed integer from P, 0 if no more
    - if Sep is #0, it won't be searched for

function GetNextItemInteger(var P: PUTF8Char; Sep: AnsiChar=',,'): PtrInt;
    Return next CSV string as signed integer from P, 0 if no more
    - if Sep is #0, it won't be searched for

function GetNextItemQWord(var P: PUTF8Char; Sep: AnsiChar=',,'): QWord;
    Return next CSV string as 64-bit unsigned integer from P, 0 if no more
    - if Sep is #0, it won't be searched for

procedure GetNextItemShortString(var P: PUTF8Char; out Dest: ShortString; Sep: AnsiChar=',,');
    Return next CSV string from P, nil if no more
    - output text would be trimmed from any left or right space
function GetNextItemString(var P: PChar; Sep: Char= ','): string;
  Return next CSV string from P, nil if no more
  - this function returns the generic string type of the compiler, and therefore can be used with
    ready to be displayed text (e.g. for the VCL)

function GetNextItemToVariant(var P: PUTF8Char; out Value: Variant; Sep: AnsiChar= ','; AllowDouble: boolean=true): boolean;
  Convert the next CSV item from an UTF-8 encoded text buffer into a variant number or RawUTF8
  varString
  - first try with GetNumericVariantFromJSON(), then fallback to RawUTF8ToVariant
  - is a wrapper around GetNextItem() + TextToVariant()

procedure GetNextItemTrimed(var P: PUTF8Char; Sep: AnsiChar; var result: RawUTF8);
  Return trimmed next CSV string from P
  - P=nil after call when end of text is reached

procedure GetNextItemTrimedCRLF(var P: PUTF8Char; var result: RawUTF8);
  Return next CRLF separated value string from P, ending #10 or #13#10 trimmed
  - any kind of line feed (CRLF or LF) will be handled, on all operating systems
  - as used e.g. by TSynNameValue.InitFromCSV and TDocVariantData.InitCSV
  - P=nil after call when end of text is reached

function GetNextLine(source: PUTF8Char; out next: PUTF8Char; andtrim: boolean=false): RawUTF8;
  Extract a line from source array of chars
  - next will contain the beginning of next line, or nil if source if ended

function GetNextStringLineToRawUnicode(var P: PChar): RawUnicode;
  Return next string delimited with #13#10 from P, nil if no more
  - this function returns a RawUnicode string type

function GetNextTChar64(var P: PUTF8Char; Sep: AnsiChar; out Buf: TChar64): PtrInt;
  Return next CSV string from P as a #0-terminated buffer, false if no more
  - if Sep is #0, will copy all characters until next whitespace char
  - returns the number of bytes stored into Buf[]

function GetNextUTF8Upper(var U: PUTF8Char): PtrUInt;
  Retrieve the next UCS4 value stored in U, then update the U pointer
  - this function will decode the UTF-8 content before using NormToUpper[]
  - will return '?' if the UCS4 value is higher than #255: so use this function only if you need to deal
    with ASCII characters (e.g. it's used for Soundex and for ContainsUTF8 function)

function GetNumericVariantFromJSON(JSON: PUTF8Char; var Value: TVarData; AllowVarDouble: boolean): boolean;
  Low-level function to set a numerical variant from an unescaped JSON number
  - returns TRUE if TextToVariantNumberType/TextToVariantNumberTypeNoDouble(JSON)
    identified it as a number and set Value to the corresponding content
  - returns FALSE if JSON is a string, or null/true/false
function GetPublishedMethods(Instance: TObject; out Methods: TPublishedMethodInfoDynArray; aClass: TClass = nil): integer;

*Retrieve published methods information about any class instance*
- will optionally accept a Class, in this case Instance is ignored
- will work with FPC and Delphi RTTI

function GetQWord(P: PUTF8Char; var err: integer): QWord;

*Get the 64-bit unsigned integer value stored in P^*
- set the err content to the index of any faulty character, 0 if conversion was successful (same as the standard val function)

function GetSectionContent(SectionFirstLine: PUTF8Char): RawUTF8; overload;

*Retrieve the whole content of a section as a string*
- SectionFirstLine may have been obtained by FindSectionFirstLine() function above

function GetSectionContent(const Content, SectionName: RawUTF8): RawUTF8; overload;

*Retrieve the whole content of a section as a string*
- use SectionFirstLine() then previous GetSectionContent()

function GetSetBaseEnum(aTypeInfo: pointer): pointer;

*Low-level helper to retrieve the base enumeration RTTI of a given set*

function GetSetName(aTypeInfo: pointer; const value): RawUTF8;

*Helper to retrieve the CSV text of all enumerate items defined in a set*
- you'd better use RTTI related classes of mORMot.pas unit, e.g. TEnumType

procedure GetSetNameShort(aTypeInfo: pointer; const value; out result: ShortString; trimlowercase: boolean=false);

*Helper to retrieve the CSV text of all enumerate items defined in a set*
- you'd better use RTTI related classes of mORMot.pas unit, e.g. TEnumType

function GetSetNameValue(aTypeInfo: pointer; var P: PUTF8Char; out EndOfObject: AnsiChar): cardinal;

*Helper to retrieve the bit mapped integer value of a set from its JSON text*
- if supplied P^ is a JSON integer number, will read it directly
- if P^ maps some ["item1","item2"] content, would fill all matching bits
- if P^ contains ["*"], would fill all bits
- returns P=nil if reached prematurely the end of content, or returns the value separator (e.g. , or }) in EndOfObject (like GetJsonField)

function GetSystemPath(kind: TSystemPath): TFileName;

*Returns an operating system folder*
- will return the full path of a given kind of private or shared folder, depending on the underlying operating system
- will use SHGetFolderPath and the corresponding CSIDL constant under Windows
- under POSIX, will return $TMP/$TMPDIR folder for spTempFolder, ~/.cache/appname for spUserData, /var/log for spLog, or the $HOME folder
- returned folder name contains the trailing path delimiter (\ or /)

function GetUnQuoteCSVItem(P: PUTF8Char; Index: PtrUInt; Sep: AnsiChar=''; Quote: AnsiChar='"'): RawUTF8; overload;

*Return n-th indexed CSV string (unquoted if needed) in P, starting at Index=0 for first one*
function GetUTF8Char(P: PUTF8Char): cardinal;

Get the WideChar stored in P^ (decode UTF-8 if necessary)
- any surrogate (UCS4>$ffff) will be returned as '?'

function GotoEndJSONItem(P: PUTF8Char; strict: boolean=false): PUTF8Char;

Reach positon just after the current JSON item in the supplied UTF-8 buffer
- buffer can be either any JSON item, i.e. a string, a number or even a JSON array (ending with ])
- a JSON object (ending with })
- returns nil if the specified buffer is not valid JSON content
- returns the position in buffer just after the item excluding the separator character - i.e. result^ may be '','}','}']

procedure GetVariantFromJSON(JSON: PUTF8Char; wasString: Boolean; var Value: variant;
TryCustomVariants: PDocVariantOptions=nil; AllowDouble: boolean=false);

Low-level function to set a variant from an unescaped JSON number or string
- expect the JSON input buffer to be already unescaped, e.g. by GetJSONField()
- is called e.g. by function VariantLoadJSON()
- will instantiate either a null, boolean, Integer, Int64, currency, double (if AllowDouble is true or
dvoAllowDoubleValue is in TryCustomVariants^) or string value (as RawUTF8), guessing the best
numeric type according to the textual content, and string in all other cases, except if
TryCustomVariants points to some options (e.g. @JSON_OPTIONS[true] for fast instance) and
input is a known object or array, either encoded as strict-JSON (i.e. {...} or [...]'), or with some
extended (e.g. BSON) syntax

function GetVariantFromNotStringJSON(JSON: PUTF8Char; var Value: TVarData;
AllowDouble: boolean): boolean;

Low-level function to set a variant from an unescaped JSON non string
- expect the JSON input buffer to be already unescaped, e.g. by GetJSONField(), and having
returned wasString=TRUE (i.e. not surrounded by double quotes)
- is called e.g. by function GetVariantFromJSON()
- will recognize null, boolean, Integer, Int64, currency, double (if AllowDouble is true) input, then
set Value and return TRUE
- returns FALSE if the supplied input has no expected JSON format

procedure GlobalLock;

Enter a giant lock for thread-safe shared process
- shall be protected as such:
GlobalLock;
try
.... do something thread-safe but as short as possible
finally
GlobalUnLock;
end;

- you should better not use such a giant-lock, but an instance-dedicated critical section - these
functions are just here to be convenient, for non time-critical process

procedure GlobalUnLock;

Release the giant lock for thread-safe shared process
- you should better not use such a giant-lock, but an instance-dedicated critical section - these
functions are just here to be convenient, for non time-critical process

procedure GetUTF8Char(P: PUTF8Char): cardinal;

Get the WideChar stored in P^ (decode UTF-8 if necessary)
- any surrogate (UCS4>$ffff) will be returned as '?'

function GotoEndJSONItem(P: PUTF8Char; strict: boolean=false): PUTF8Char;

Reach positon just after the current JSON item in the supplied UTF-8 buffer
- buffer can be either any JSON item, i.e. a string, a number or even a JSON array (ending with ])
- a JSON object (ending with })
- returns nil if the specified buffer is not valid JSON content
- returns the position in buffer just after the item excluding the separator character - i.e. result^ may be '','}','}']

function GetUTF8Char(P: PUTF8Char): cardinal;

Get the WideChar stored in P^ (decode UTF-8 if necessary)
- any surrogate (UCS4>$ffff) will be returned as '?'

function GotoEndJSONItem(P: PUTF8Char; strict: boolean=false): PUTF8Char;

Reach positon just after the current JSON item in the supplied UTF-8 buffer
- buffer can be either any JSON item, i.e. a string, a number or even a JSON array (ending with ])
- a JSON object (ending with })
- returns nil if the specified buffer is not valid JSON content
- returns the position in buffer just after the item excluding the separator character - i.e. result^ may be '','}','}']
function GotoEndOfJSONString(P: PUTF8Char): PUTF8Char;
   \* Get the next character after a quoted buffer \*
   - the first character in P^ must be "
   - it will return the latest " position, ignoring \" within

function GotoEndOfQuotedString(P: PUTF8Char): PUTF8Char;
   \* Get the next character after a quoted buffer \*
   - the first character in P^ must be either ', either "
   - it will return the latest quote position, ignoring double quotes within

function GotoNextJSONItem(P: PUTF8Char; NumberOfItemsToJump: cardinal=1;
   EndOfObject: PAnsiChar=nil): PUTF8Char;
   \* Reach the position of the next JSON item in the supplied UTF-8 buffer \*
   - buffer can be either any JSON item, i.e. a string, a number or even a JSON array (ending with ]) or a JSON object (ending with }
   - returns nil if the specified number of items is not available in buffer
   - returns the position in buffer after the item including the separator character (optionally in EndOfObject) - i.e. result will be at the start of the next object, and EndOfObject may be ',','}',']

function GotoNextJSONObjectOrArray(P: PUTF8Char; EndChar: AnsiChar): PUTF8Char; overload;
   \* Reach the position of the next JSON object of JSON array \*
   - first char is expected to be just after the initial '[' or '{'
   - specify '] or '}' as the expected EndChar
   - will return nil in case of parsing error or unexpected end (#0)
   - will return the next character after ending [ or } - i.e. may be , ]

function GotoNextJSONObjectOrArray(P: PUTF8Char): PUTF8Char; overload;
   \* Reach the position of the next JSON object of JSON array \*
   - first char is expected to be either '[' or '{'
   - will return nil in case of parsing error or unexpected end (#0)
   - will return the next character after ending [ or } - i.e. may be , ]

function GotoNextJSONObjectOrArrayMax(P,PMax: PUTF8Char): PUTF8Char;
   \* Reach the position of the next JSON object of JSON array \*
   - first char is expected to be either '[' or '{'
   - this version expects a maximum position in PMax: it may be handy to break the parsing for HUGE content - used e.g. by JSONArrayCount(P,PMax)
   - will return nil in case of parsing error or if P reached PMax limit
   - will return the next character after ending [ or } - i.e. may be , ]

function GotoNextJSONPropName(P: PUTF8Char): PUTF8Char;
   \* Read the position of the JSON value just after a property identifier \*
   - this function will handle strict JSON property name (i.e. a "string"), but also MongoDB extended syntax, e.g. {age:{$gt:18}} or {'people.age':{$gt:18}} see @http://docs.mongodb.org/manual/reference/mongodb-extended-json

function GotoNextLine(source: PUTF8Char): PUTF8Char;
   \* Fast go to next text line, ended by #13 or #13#10 \*
   - returns the beginning of next line, or nil if source^=#0 was reached

function GotoNextNotSpace(P: PUTF8Char): PUTF8Char;
   \* Get the next character not in [#1..'] \*
function GotoNextNotSpaceSameLine(P: PUTF8Char): PUTF8Char;
Get the next character not in [#9,'']

function GotoNextSpace(P: PUTF8Char): PUTF8Char;
Get the next character in [#1.'']

function GotoNextVarInt(Source: PByte): pointer;
Jump a value in the 32-bit or 64-bit variable-length integer buffer

function GotoNextVarString(Source: PByte): pointer;
Jump a value in variable-length text buffer

function GUIDToRawUTF8(const guid: TGUID): RawUTF8;
Convert a TGUID into UTF-8 encoded text
- will return e.g. '{3F2504E0-4F89-11D3-9A0C-0305E82C3301}' (with the {})
- if you do not need the embracing {}, use ToUTF8() overloaded function

function GUIDToShort(const guid: TGUID): TGUIDShortString; overload;
Convert a TGUID into text
- will return e.g. '{3F2504E0-4F89-11D3-9A0C-0305E82C3301}' (with the {})
- using a shortstring will allow fast allocation on the stack, so is preferred e.g. when providing a GUID to a ESynException.CreateUTF8()

procedure GUIDToShort(const guid: TGUID; out dest: TGUIDShortString); overload;
Convert a TGUID into text
- will return e.g. '{3F2504E0-4F89-11D3-9A0C-0305E82C3301}' (with the {})
- using a shortstring will allow fast allocation on the stack, so is preferred e.g. when providing a GUID to a ESynException.CreateUTF8()

function GUIDToString(const guid: TGUID): string;
Convert a TGUID into text
- will return e.g. '{3F2504E0-4F89-11D3-9A0C-0305E82C3301}' (with the {})
- this version is faster than the one supplied by SysUtils

function GUIDToText(P: PUTF8Char; guid: PByteArray): PUTF8Char;
Append a TGUID binary content as text
- will store e.g. '3F2504E0-4F89-11D3-9A0C-0305E82C3301' (without any {})
- this will be the format used for JSON encoding, e.g.
  { "UID": "C9A646D3-9C61-4CB7-BFCD-EE2522C8F633" } 

function Hash128(const Elem; Hasher: THasher): cardinal;
Hash one THash128 value with the supplied Hasher function

function Hash128Index(P: PHash128Rec; Count: integer; h: PHash128Rec): integer;
Fast O(n) search of a 128-bit item in an array of such values

function Hash256(const Elem; Hasher: THasher): cardinal;
Hash one THash256 value with the supplied Hasher function

function Hash256Index(P: PHash256Rec; Count: integer; h: PHash256Rec): integer; overload;
Fast O(n) search of a 256-bit item in an array of such values
function Hash32(Data: PCardinalArray; Len: integer): cardinal; overload;

Our custom efficient 32-bit hash/checksum function
- a Fletcher-like checksum algorithm, not a hash function: has less collisions than Adler32 for short strings, but more than xxhash32 or crc32/crc32c
- written in simple plain pascal, with no L1 CPU cache pollution, but we also provide optimized x86/x64 assembly versions, since the algorithm is used heavily e.g. for TDynArray binary serialization, TSQLRestStorageInMemory binary persistence, or CompressSynLZ/StreamSynLZ/FileSynLZ
- some numbers on Linux x86_64:
  2500 hash32 in 707us i.e. 3536067/s or 7.3 GB/s
  2500 xxhash32 in 1.34ms i.e. 1861504/s or 3.8 GB/s
  2500 crc32c in 943us i.e. 2651113/s or 5.5 GB/s (SSE4.2 disabled)
  2500 crc32c in 387us i.e. 6459948/s or 13.4 GB/s (SSE4.2 enabled)

function Hash32(const Text: RawByteString): cardinal; overload;

Our custom efficient 32-bit hash/checksum function
- a Fletcher-like checksum algorithm, not a hash function: has less collisions than Adler32 for short strings, but more than xxhash32 or crc32/crc32c
- overloaded function using RawByteString for binary content hashing, whatever the codepage is

function Hash512(const Elem; Hasher: THasher): cardinal;
Hash one THash512 value with the supplied Hasher() function

function HashAnsiString(const Elem; Hasher: THasher): cardinal;
Hash one AnsiString content with the supplied Hasher() function

function HashAnsiStringI(const Elem; Hasher: THasher): cardinal;
Case-insensitive hash one AnsiString content with the supplied Hasher() function

function HashByte(const Elem; Hasher: THasher): cardinal;
Hash one Byte value

function HashFile(const FileName: TFileName; Hasher: THasher=nil): cardinal;
Compute the 32-bit default hash of a file content
- you can specify your own hashing function if DefaultHasher is not what you expect

function HashInt64(const Elem; Hasher: THasher): cardinal;
Hash one Int64/Qword value with the supplied Hasher() function

function HashInteger(const Elem; Hasher: THasher): cardinal;
Hash one Integer/cardinal value - simply return the value ignore Hasher() parameter

function HashPointer(const Elem; Hasher: THasher): cardinal;
Hash one pointer value with the supplied Hasher() function
- this version is not the same as HashPtrUInt, since it will always use the hasher function

function HashPtrUInt(const Elem; Hasher: THasher): cardinal;
Hash one PtrUInt (=NativeUInt) value with the supplied Hasher() function

function HashSynUnicode(const Elem; Hasher: THasher): cardinal;
Hash one SynUnicode content with the supplied Hasher() function
- work with WideString for all Delphi versions, or UnicodeString in Delphi 2009+
function HashSynUnicodeI(const Elem; Hasher: THasher): cardinal;
Case-insensitive hash one SynUnicode content with the supplied Hasher() function
- work with WideString for all Delphi versions, or UnicodeString in Delphi 2009+

function HashVariant(const Elem; Hasher: THasher): cardinal;
Case-sensitive hash one variant content with the supplied Hasher() function

function HashVariantI(const Elem; Hasher: THasher): cardinal;
Case-insensitive hash one variant content with the supplied Hasher() function

function HashWideString(const Elem; Hasher: THasher): cardinal;
Hash one WideString content with the supplied Hasher() function
- work with WideString for all Delphi versions

function HashWideStringI(const Elem; Hasher: THasher): cardinal;
Case-insensitive hash one WideString content with the supplied Hasher() function
- work with WideString for all Delphi versions

function HashWord(const Elem; Hasher: THasher): cardinal;
Hash one Word value

function HexDisplayToBin(Hex: PAnsiChar; Bin: PByte; BinBytes: integer): boolean;
Fast conversion from hexa chars into a binary buffer

function HexDisplayToCardinal(Hex: PAnsiChar; out aValue: cardinal): boolean;
Fast conversion from hexa chars into a cardinal
- reverse function of CardinalToHex()
  - returns false and set aValue=0 if Hex is not a valid hexadecimal 32-bit unsigned integer
  - returns true and set aValue with the decoded number, on success

function HexDisplayToInt64(const Hex: RawByteString): Int64; overload;
Fast conversion from hexa chars into a cardinal
- reverse function of Int64ToHex()
  - returns 0 if the supplied text buffer is not a valid hexadecimal 64-bit signed integer

function HexDisplayToInt64(Hex: PAnsiChar; out aValue: Int64): boolean; overload;
Inline gives an error under release conditions with FPC fast conversion from hexa chars into a cardinal
- reverse function of Int64ToHex()
  - returns false and set aValue=0 if Hex is not a valid hexadecimal 64-bit signed integer
  - returns true and set aValue with the decoded number, on success

function HexToBin(const Hex: RawUTF8): RawByteString; overload;
Fast conversion from hexa chars into binary data

function HexToBin(Hex: PAnsiChar; Bin: PByte; BinBytes: Integer): boolean; overload;
Fast conversion from hexa chars into binary data
- BinBytes contain the bytes count to be converted: Hex^ must contain at least BinBytes*2 chars to be converted, and Bin^ enough space
  - if Bin=nil, no output data is written, but the Hex^ format is checked
  - return false if any invalid (non hexa) char is found in Hex^ and
  - using this function with Bin^ as an integer value will decode in big-endian order
    (most-significant byte first)
<table>
<thead>
<tr>
<th>Procedure Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HexToBinFast</strong> (Hex: PAnsiChar; Bin: PByte; BinBytes: Integer);</td>
<td>Fast conversion with no validity check from hexa chars into binary data</td>
</tr>
<tr>
<td><strong>HexToChar</strong> (Hex: PAnsiChar; Bin: PUTF8Char): boolean;</td>
<td>Fast conversion from one hexa char pair into a 8 bit AnsiChar - return false if any invalid (non hexa) char is found in Hex^ - similar to HexToBin(Hex,Bin,1) but with Bin&lt;&gt;nil - use HexToCharValid if you want to check a hexadecimal char content</td>
</tr>
<tr>
<td><strong>HexToCharValid</strong> (Hex: PAnsiChar): boolean;</td>
<td>Fast conversion from one hexa char pair into a 8 bit AnsiChar - return false if any invalid (non hexa) char is found in Hex^ - similar to HexToBin(Hex,nil,1)</td>
</tr>
<tr>
<td><strong>HexToWideChar</strong> (Hex: PAnsiChar): cardinal;</td>
<td>Fast conversion from two hexa bytes into a 16 bit UTF-16 WideChar - similar to HexToBin(Hex,@wordvar,2) + bswap(wordvar)</td>
</tr>
<tr>
<td><strong>IdemFileExt</strong> (p: PUTF8Char; extup: PAnsiChar; sepChar: AnsiChar='.'): Boolean;</td>
<td>Returns true if the file name extension contained in p^ is the same same as extup^ - ignore case - extup^ must be already Upper - chars are compared as WinAnsi (codepage 1252), not as UTF-8 - could be used e.g. like IdemFileExt(aFileName,'.JP');</td>
</tr>
<tr>
<td><strong>IdemFileExts</strong> (p: PUTF8Char; const extup: array of PAnsiChar; sepChar: AnsiChar='.'): integer;</td>
<td>Returns matching file name extension index as extup^ - ignore case - extup[] must be already Upper - chars are compared as WinAnsi (codepage 1252), not as UTF-8 - could be used e.g. like IdemFileExts(aFileName,['.PAS','.INC']);</td>
</tr>
<tr>
<td><strong>IdemPChar</strong> (p: PUTF8Char; up: PAnsiChar): boolean;</td>
<td>Returns true if the beginning of p^ is the same as up^ - ignore case - up^ must be already Upper - chars are compared as 7 bit Ansi only (no accented characters): but when you only need to search for field names e.g. IdemPChar() is prefered, because it'll be faster than IdemPCharU(), if UTF-8 decoding is not mandatory - if p is nil, will return FALSE - if up is nil, will return TRUE</td>
</tr>
<tr>
<td><strong>IdemPCharAndGetNextItem</strong> (var source: PUTF8Char; const searchUp: RawUTF8; var Item: RawUTF8; Sep: AnsiChar=#13): boolean;</td>
<td>Return true if IdemPChar(source,searchUp), and retrieve the value item - typical use may be: if IdemPCharAndGetNextItem(P, 'CONTENT-DISPOSITION: FORM-DATA; NAME=&quot;',Name,'&quot;,') then ...</td>
</tr>
<tr>
<td><strong>IdemPCharAndGetNextLine</strong> (var source: PUTF8Char; searchUp: PAnsiChar): boolean;</td>
<td>Return true if IdemPChar(source,searchUp), and go to the next line of source</td>
</tr>
</tbody>
</table>
function IdemPCharArray(p: PUTF8Char; const upArrayBy2Chars: RawUTF8): integer;
overload;

Returns the index of a matching beginning of p^ in upArray two characters
- returns -1 if no item matched
- ignore case - upArray^ must be already Upper
- chars are compared as 7 bit Ansi only (no accentuated characters)

function IdemPCharArray(p: PUTF8Char; const upArray: array of PAnsiChar): integer;
overload;

Returns the index of a matching beginning of p^ in upArray[]
- returns -1 if no item matched
- ignore case - upArray^ must be already Upper
- chars are compared as 7 bit Ansi only (no accentuated characters)
- warning: this function expects upArray[] items to have AT LEAST TWO CHARS (it will use a fast comparison of initial 2 bytes)

function IdemPCharU(p, up: PUTF8Char): boolean;

Returns true if the beginning of p^ is the same as up^
- ignore case - up^ must be already Upper
- this version will decode the UTF-8 content before using NormToUpper[], so it will be slower than the IdemPChar() function above, but will handle WinAnsi accentuated characters (e.g. 'é' acute will be matched as 'E')

function IdemPCharW(p: PWideChar; up: PUTF8Char): boolean;

Returns true if the beginning of p^ is same as up^
- ignore case - up^ must be already Upper
- this version expects p^ to point to an Unicode char array

function IdemPCharWithoutWhiteSpace(p: PUTF8Char; up: PAnsiChar): boolean;

Returns true if the beginning of p^ is the same as up^, ignoring white spaces
- ignore case - up^ must be already Upper
- any white space in the input p^ buffer is just ignored
- chars are compared as 7 bit Ansi only (no accentuated characters): but when you only need to search for field names e.g. IdemPChar() is preferred, because it'll be faster than IdemPCharU(), if UTF-8 decoding is not mandatory
- if p is nil, will return FALSE
- if up is nil, will return TRUE

function IdemPropName(P1,P2: PUTF8Char; P1Len,P2Len: PtrInt): boolean; overload;

Case insensitive comparison of ASCII identifiers
- use it with property names values (i.e. only including A..Z,0..9, _ chars)
- this version expects P1 and P2 to be a PAnsiChar with specified lengths

function IdemPropName(const P1: shortstring; P2: PUTF8Char; P2Len: PtrInt): boolean; overload;

Case insensitive comparison of ASCII identifiers
- use it with property names values (i.e. only including A..Z,0..9, _ chars)
- this version expects P2 to be a PAnsiChar with a specified length

function IdemPropName(const P1,P2: shortstring): boolean; overload;

Case insensitive comparison of ASCII identifiers
- use it with property names values (i.e. only including A..Z,0..9, _ chars)
function IdemPropNameU(const P1, P2: RawUTF8): boolean; overload;
  Case insensitive comparison of ASCII identifiers
  - use it with property names values (i.e. only including A..Z,0..9,_,chars)

function IdemPropNameU(const P1: RawUTF8; P2: PUTF8Char; P2Len: PTrint): boolean; overload;
  Case insensitive comparison of ASCII identifiers
  - use it with property names values (i.e. only including A..Z,0..9,_,chars)
  - this version expects P2 to be a PAnsiChar with specified length

function IdemPropNameUSameLen(P1, P2: PUTF8Char; P1P2Len: PTrint): boolean;
  Case insensitive comparison of ASCII identifiers of same length
  - use it with property names values (i.e. only including A..Z,0..9,_,chars)
  - this version expects P1 and P2 to be a PAnsiChar with an already checked identical length, so
    may be used for a faster process, e.g. in a loop
  - if P1 and P2 are RawUTF8, you should better call overloaded function IdemPropNameU(const
    P1, P2: RawUTF8), which would be slightly faster by using the length stored before the actual text
    buffer of each RawUTF8

procedure IncludeInt64(var Values, Included: TInt64DynArray; IncludedSortSize: Integer=32);
  Ensure some 64-bit integer from Values[] will only contain Included[]
  - Included is declared as var, since it will be sorted in-place during process if it contains more than
    IncludedSortSize items (i.e. if the sort is worth it)

procedure IncludeInteger(var Values, Included: TIntegerDynArray; IncludedSortSize: Integer=32);
  Ensure some 32-bit integer from Values[] will only contain Included[]
  - Included is declared as var, since it will be sorted in-place during process if it contains more than
    IncludedSortSize items (i.e. if the sort is worth it)

function IncludeTrailingURIDelimiter(const URI: RawByteString): RawByteString;
  Ensure the supplied URI contains a trailing '/' character

procedure InitializeCriticalSectionIfNeededAndEnter(var CS: TRTLCriticalSection);
  On need initialization of a mutex, then enter the lock
  - if the supplied mutex has been initialized, do nothing
  - if the supplied mutex is void (i.e. all filled with 0), initialize it

function InsertInteger(var Values: TIntegerDynArray; var ValuesCount: integer; Value: Integer;
  Index: PtrInt; CoValues: PIntegerDynArray=nil): PtrInt;
  Insert an integer value at the specified index position of a dynamic array of integers
  - if Index is invalid, the Value is inserted at the end of the array

function Int18ToChars3(Value: cardinal): RawUTF8; overload;
  Compute the value as encoded by TTextWriter.AddInt18ToChars3() method

procedure Int18ToChars3(Value: cardinal; var result: RawUTF8); overload;
  Compute the value as encoded by TTextWriter.AddInt18ToChars3() method

function Int32ToUtf8(Value: PtrInt): RawUTF8; overload;
  Use our fast RawUTF8 version of IntToStr()
  - without any slow UnicodeString=String->AnsiString conversion for Delphi 2009
  - only useful if our Enhanced Runtime (or LVCL) library is not installed
procedure Int32ToUTF8(Value: PointerInt; var result: RawUTF8); overload;

Use our fast RawUTF8 version of IntToStr()
- without any slow UnicodeString=String->AnsiString conversion for Delphi 2009
- result as var parameter saves a local assignment and a try..finally

function Int64DynArrayToCSV(const Values: TInt64DynArray; const Prefix: RawUTF8=''; const Suffix: RawUTF8=''; InlinedValue: boolean=false): RawUTF8; overload;

Return the corresponding CSV text from a dynamic array of 64-bit integers
- you can set some custom Prefix and Suffix text

function Int64DynArrayToCSV(const Values: PBInt64Array; const Prefix: RawUTF8=''; const Suffix: RawUTF8=''; InlinedValue: boolean=false): RawUTF8; overload;

Return the corresponding CSV text from a dynamic array of 64-bit integers
- you can set some custom Prefix and Suffix text

function Int64Scan(P: PBInt64Array; Count: PointerInt; const Value: Int64): PBInt64;

Fast search of an integer position in a 64-bit integer array
- Count is the number of Int64 entries in P^,
- returns P where P^=Value
- returns nil if Value was not found

function Int64ScanExists(P: PBInt64Array; Count: PointerInt; const Value: Int64): boolean;

Fast search of an integer value in a 64-bit integer array
- returns true if P^=Value within Count entries
- returns false if Value was not found

function Int64ScanIndex(P: PBInt64Array; Count: PointerInt; const Value: Int64): PointerInt;

Fast search of an integer position in a signed 64-bit integer array
- Count is the number of Int64 entries in P^,
- returns index of P^[index]=Value
- returns -1 if Value was not found

procedure Int64ToHex(aInt64: Int64; var result: RawUTF8); overload;

Fast conversion from a Int64 value into hexa chars, ready to be displayed
- use internally BinToHexDisplay()
- reverse function of HexDisplayToInt64()

function Int64ToHex(aInt64: Int64): RawUTF8; overload;

Fast conversion from a Int64 value into hexa chars, ready to be displayed
- use internally BinToHexDisplay()
- reverse function of HexDisplayToInt64()

procedure Int64ToHexShort(aInt64: Int64; out result: TShort16); overload;

Fast conversion from a Int64 value into hexa chars, ready to be displayed
- such result type would avoid a string allocation on heap

function Int64ToHexShort(aInt64: Int64): TShort16; overload;

Fast conversion from a Int64 value into hexa chars, ready to be displayed
- such result type would avoid a string allocation on heap
function Int64ToHexString(aInt64: Int64): string;
  Fast conversion from a Int64 value into hexa chars, ready to be displayed
  - use internally BinToHexDisplay()
  - reverse function of HexDisplayToInt64()

procedure Int64ToUInt32(Values64: PInt64Array; Values32: PCardinalArray; Count: PInt);
  Copy some Int64 values into an unsigned integer array

function Int64ToUtf8(Value: Int64): RawUTF8; overload;
  Use our fast RawUTF8 version of IntToStr()
  - without any slow UnicodeString=String->AnsiString conversion for Delphi 2009
  - only useful if our Enhanced Runtime (or LVCL) library is not installed

procedure Int64ToUtf8(Value: Int64; var result: RawUTF8); overload;
  Use our fast RawUTF8 version of IntToStr()
  - without any slow UnicodeString=String->AnsiString conversion for Delphi 2009
  - result as var parameter saves a local assignment and a try..finally

function IntegerDynArrayLoadFrom(Source: PAnsiChar; var Count: integer; NoHash32Check: boolean=false): PIntegerArray;
  Wrap an Integer dynamic array BLOB content as stored by TDynArray.SaveTo
  - same as TDynArray.LoadFrom() with no memory allocation nor memory copy: so is much faster
    than creating a temporary dynamic array to load the data
  - will return nil if no or invalid data, or a pointer to the integer array otherwise, with the items
    number stored in Count
  - slightly faster than SimpleDynArrayLoadFrom(Source,TypeInfo(TIntegerDynArray),Count)

function IntegerDynArrayToCSV(const Values: TIntegerDynArray; const Prefix: RawUTF8=''; const Suffix: RawUTF8=''; InlinedValue: boolean=false): RawUTF8;
  overload;
  Return the corresponding CSV text from a dynamic array of 32-bit integer
  - you can set some custom Prefix and Suffix text

function IntegerDynArrayToCSV(Values: PIntegerArray; ValuesCount: integer; const Prefix: RawUTF8=''; const Suffix: RawUTF8=''; InlinedValue: boolean=false): RawUTF8;
  overload;
  Return the corresponding CSV text from a dynamic array of 32-bit integer
  - you can set some custom Prefix and Suffix text

function IntegerScan(P: PCardinalArray; Count: PInt; Value: cardinal): PCardinal;
  Fast search of an unsigned integer position in an integer array
  - Count is the number of cardinal entries in P^
  - returns P where P^=Value
  - returns nil if Value was not found

function IntegerScanExists(P: PCardinalArray; Count: PInt; Value: cardinal): boolean;
  Fast search of an unsigned integer in an integer array
  - returns true if P^=Value within Count entries
  - returns false if Value was not found
function IntegerScanIndex(P: PCardinalArray; Count: PtrInt; Value: cardinal): PtrInt;
  *Fast search of an unsigned integer position in an integer array*
  - Count is the number of integer entries in P
  - return index of P^[index]=Value
  - return -1 if Value was not found

function InterfaceArrayAdd(var aInterfaceArray; const aItem: IUnknown): PtrInt;
  *Wrapper to add an item to a T*InterfaceArray* dynamic array storage

procedure InterfaceArrayAddOnce(var aInterfaceArray; const aItem: IUnknown);
  *Wrapper to add once an item to a T*InterfaceArray* dynamic array storage

procedure InterfaceArrayDelete(var aInterfaceArray; aItemIndex: PtrInt); overload;
  *Wrapper to delete an item in a T*InterfaceArray* dynamic array storage
  - do nothing if the item is not found in the dynamic array

function InterfaceArrayDelete(var aInterfaceArray; const aItem: IUnknown): PtrInt;
  overload;
  *Wrapper to delete an item in a T*InterfaceArray* dynamic array storage
  - search is performed by address/reference, not by content
  - do nothing if the item is not found in the dynamic array

function InterfaceArrayFind(const aInterfaceArray; const aItem: IUnknown): PtrInt;
  *Wrapper to search an item in a T*InterfaceArray* dynamic array storage
  - search is performed by address/reference, not by content
  - return -1 if the item is not found in the dynamic array, or the index of the matching entry otherwise

function InterlockedDecrement(var I: Integer): Integer;
  *Compatibility function, to be implemented according to the running CPU*
  - expect the same result as the homonymous Win32 API function

function InterlockedIncrement(var I: Integer): Integer;
  *Compatibility function, to be implemented according to the running CPU*
  - expect the same result as the homonymous Win32 API function

function IntervalTextToDateTim(Text: PUTF8Char): TDateTime;
  *Interval date/time conversion from simple text*
  - expected format does not match ISO-8601 Time intervals format, but Oracle interval literal representation, i.e. '+/-D HH:MM:SS'
  - e.g. IntervalTextToDateTim('+0 06:03:20') will return 0.25231481481 and IntervalTextToDateTim('-20 06:03:20') -20.252314815
  - as a consequence, negative intervals will be written as TDateTime values:
    - DateTimeToIso8601Text(IntervalTextToDateTim('+0 06:03:20'))='1899-12-31T06:03:20'
    - DateTimeToIso8601Text(IntervalTextToDateTim('-1 06:03:20'))='1899-12-31T06:03:20'
    - DateTimeToIso8601Text(IntervalTextToDateTim('-2 06:03:20'))='1899-12-31T06:03:20'

procedure IntervalTextToDateTimeVar(Text: PUTF8Char; var result: TDateTime);
  *Interval date/time conversion from simple text*
  - expected format does not match ISO-8601 Time intervals format, but Oracle interval literal representation, i.e. '+/-D HH:MM:SS'
  - e.g. '+1 06:03:20' will return 1.25231481481
function IntToString(Value: cardinal): string; overload;
    Faster version than default SysUtils.IntToStr implementation

function IntToString(Value: integer): string; overload;
    Faster version than default SysUtils.IntToStr implementation

function IntToString(Value: Int64): string; overload;
    Faster version than default SysUtils.IntToStr implementation

function IntToThousandString(Value: integer; const ThousandSep: TShort4='‚','): shortstring;
    Convert an integer value into its textual representation with thousands marked
    - ThousandSep is the character used to separate thousands in numbers with more than three
digits to the left of the decimal separator

function IP4Text(ip4: cardinal): shortstring; overload;
    Convert a 32-bit integer (storing a IP4 address) into its full notation
    - returns e.g. '1.2.3.4' for any valid address, or '' if ip4=0

function IP6Text(ip6: PHash128): shortstring; overload;
    Convert a 128-bit buffer (storing an IP6 address) into its full notation
    - returns e.g. '2001:0db8:0a0b:12f0:0000:0000:0000:0001'

procedure IP6Text(ip6: PHash128; result: PShortString); overload;
    Convert a 128-bit buffer (storing an IP6 address) into its full notation
    - returns e.g. '2001:0db8:0a0b:12f0:0000:0000:0000:0001'

function IsAnsiCompatible(PC: PAnsiChar): boolean; overload;
    Return TRUE if the supplied buffer only contains 7-bits Ansi characters

function IsAnsiCompatible(PC: PAnsiChar; Len: PtrUInt): boolean; overload;
    Return TRUE if the supplied buffer only contains 7-bits Ansi characters

function IsAnsiCompatible(const Text: RawByteString): boolean; overload;
    Return TRUE if the supplied text only contains 7-bits Ansi characters

function IsAnsiCompatibleW(PW: PWideChar): boolean; overload;
    Return TRUE if the supplied UTF-16 buffer only contains 7-bits Ansi characters

function IsAnsiCompatibleW(PW: PWideChar; Len: PtrInt): boolean; overload;
    Return TRUE if the supplied UTF-16 buffer only contains 7-bits Ansi characters

function IsBase64(sp: PAnsiChar; len: PtrInt): boolean; overload;
    Check if the supplied text is a valid Base64 encoded stream

function IsBase64(const s: RawByteString): boolean; overload;
    Check if the supplied text is a valid Base64 encoded stream

function IsCaseSensitive(P: PUTF8Char; Plen: PtrInt): boolean; overload;
    Check if the supplied text has some case-insensitive 'a'..'z','A'..'Z' chars
    - will therefore be correct with true UTF-8 content, but only for 7 bit

function IsCaseSensitive(const S: RawUTF8): boolean; overload;
    Check if the supplied text has some case-insensitive 'a'..'z','A'..'Z' chars
    - will therefore be correct with true UTF-8 content, but only for 7 bit
function IsContentCompressed(Content: Pointer; Len:_PTRInt): boolean;

    Retrieve if some content is compressed, from a supplied binary buffer
    - returns TRUE, if the header in binary buffer "may" be compressed (this method can trigger false positives), e.g. begin with most common already compressed zip/gz/gif/png/jpeg/avi/mp3/mp4 markers (aka "magic numbers")

function IsDirectoryWritable(const Directory: TFileName): boolean;

    Check if the directory is writable for the current user
    - try to write a small file with a random name

function IsEqual(const A,B: THash160): boolean; overload;

    Returns TRUE if all 20 bytes of both 160-bit buffers do match
    - e.g. a SHA-1 digest
    - this function is not sensitive to any timing attack, so is designed for cryptographic purpose

function IsEqual(const A,B: THash256): boolean; overload;

    Returns TRUE if all 32 bytes of both 256-bit buffers do match
    - e.g. a SHA-256 digest, or a TECCSignature result
    - this function is not sensitive to any timing attack, so is designed for cryptographic purpose

function IsEqual(const A,B; count:_PTRInt): boolean; overload;

    Returns TRUE if all bytes of both buffers do match
    - this function is not sensitive to any timing attack, so is designed for cryptographic purposes - use CompareMem/CompareMemSmall/CompareMemFixed as faster alternatives for general-purpose code

function IsEqual(const A,B: THash512): boolean; overload;

    Returns TRUE if all 64 bytes of both 512-bit buffers do match
    - e.g. two SHA-512 digests
    - this function is not sensitive to any timing attack, so is designed for cryptographic purpose

function IsEqual(const A,B: THash384): boolean; overload;

    Returns TRUE if all 48 bytes of both 384-bit buffers do match
    - e.g. a SHA-384 digest
    - this function is not sensitive to any timing attack, so is designed for cryptographic purpose

function IsEqual(const A,B: THash128): boolean; overload;

    Returns TRUE if all 16 bytes of both 128-bit buffers do match
    - e.g. a MD5 digest, or an AES block
    - this function is not sensitive to any timing attack, so is designed for cryptographic purpose - and it is also branchless therefore fast

function IsEqualGUID(guid1, guid2: PGUID): Boolean; overload;

    Compare two TGUID values
    - this version is faster than the one supplied by SysUtils

function IsEqualGUID(const guid1, guid2: TGUID): Boolean; overload;

    Compare two TGUID values
    - this version is faster than the one supplied by SysUtils

function IsEqualGUIDArray(const guid: TGUID; const guides: array of TGUID): integer;

    Returns the index of a matching TGUID in an array
    - returns -1 if no item matched
    Check if a codepage should be handled by a TSynAnsiFixedWidth page

function IsHex(const Hex: RawByteString; BinBytes: integer): boolean;
    Fast check if the supplied Hex buffer is an hexadecimal representation of a binary buffer of a given number of bytes

function IsHTMLContentTypeTextual(Headers: PUTF8Char): Boolean;
    Returns TRUE if the supplied HTML Headers contains 'Content-Type: text/...', 'Content-Type: application/json' or 'Content-Type: application/xml'

function IsInitializedCriticalSection(const CS: TRTLCriticalSection): Boolean;
    Returns TRUE if the supplied mutex has been initialized
    - will check if the supplied mutex is void (i.e. all filled with 0 bytes)

function IsIso8601(P: PUTF8Char; L: integer): boolean;
    Test if P^ contains a valid ISO-8601 text encoded value
    - calls internally Iso8601ToTimeLogPUTF8Char() and returns true if contains at least a valid year (YYYY)

function IsLeapYear(Year: cardinal): boolean;
    Our own fast version of the corresponding low-level RTL function

function IsNullGUID(const guid: TGUID): Boolean;
    Check if a TGUID value contains only 0 bytes
    - this version is faster than the one supplied by SysUtils

function Iso8601CheckAndDecode(P: PUTF8Char; L: integer; var Value: TDateTime): boolean;
    Date/Time conversion from strict ISO-8601 content
    - recognize 'YYYY-MM-DDThh:mm:ss[.sss]' or 'YYYY-MM-DD' or 'Thh:mm:ss[.sss]' patterns, as e.g. generated by TTextWriter.AddDateTime() or RecordSaveJSON()
    - will also recognize '.sss' milliseconds suffix, if any

function Iso8601ToDatePUTF8Char(P: PUTF8Char; L: integer=0): TDateTime;
    Date conversion from ISO-8601 (with no Time part)
    - recognize 'YYYY-MM-DD' and 'YYYYMMDD' format into Y,M,D variables
    - if L is left to default 0, it will be computed from StrLen(P)

function Iso8601ToDateTime(const S: RawByteString): TDateTime; overload;
    Date/Time conversion from ISO-8601
    - handle 'YYYYMMDDThhmss' and 'YYYY-MM-DD hh:mm:ss' format
    - will also recognize '.sss' milliseconds suffix, if any

function Iso8601ToDateTimePUTF8Char(P: PUTF8Char; L: integer=0): TDateTime;
    Date/Time conversion from ISO-8601
    - handle 'YYYYMMDDThhmss' and 'YYYY-MM-DD hh:mm:ss' format
    - will also recognize '.sss' milliseconds suffix, if any
    - if L is left to default 0, it will be computed from StrLen(P)
procedure Iso8601ToDateTimeUTF8CharVar(P: PUTF8Char; L: integer; var result: TDateTime);

*Date/Time conversion from ISO-8601*
- handle 'YYYYMMDDThh:mm:ss' and 'YYYY-MM-DD hh:mm:ss' format, with potentially shorten versions has handled by the ISO-8601 standard (e.g. 'YYYY')
- will also recognize '.sss' milliseconds suffix, if any
- if L is left to default 0, it will be computed from StrLen(P)

function Iso8601ToDateLog(const S: RawByteString): TTimeLog;

*Convert a Iso8601 encoded string into a TTimeLog value*
- handle TTimeLog bit-encoded Int64 format
- use this function only for fast comparison between two Iso8601 date/time
- conversion is faster than Iso8601ToDateTime: use only binary integer math

function Iso8601ToDateLogUTF8Char(P: PUTF8Char; L: integer; ContainsNoTime: PBoolean=nil): TTimeLog;

*Convert a Iso8601 encoded string into a TTimeLog value*
- handle TTimeLog bit-encoded Int64 format
- use this function only for fast comparison between two Iso8601 date/time
- conversion is faster than Iso8601ToDateTime: use only binary integer math
- ContainsNoTime optional pointer can be set to a boolean, which will be set according to the layout in P (e.g. TRUE for '2012-05-26')
- returns 0 in case of invalid input string

function Iso8601ToDateUTF8Char(P: PUTF8Char; L: integer; var H,M,S,MS: cardinal); overload;

*Time conversion from ISO-8601 (with no Date part)*
- recognize 'hhmmss' and 'hh:mm:ss' format into H,M,S variables
- will also recognize '.sss' milliseconds suffix, if any, into MS
- if L is left to default 0, it will be computed from StrLen(P)

function Iso8601ToDateUTF8Char(P: PUTF8Char; L: integer=0): TDateTime; overload;

*Time conversion from ISO-8601 (with no Date part)*
- handle 'hhmmss' and 'hh:mm:ss' format
- will also recognize '.sss' milliseconds suffix, if any
- if L is left to default 0, it will be computed from StrLen(P)

procedure Iso8601ToDateUTF8CharVar(P: PUTF8Char; L: integer; var result: TDateTime);

*Time conversion from ISO-8601 (with no Date part)*
- handle 'hhmmss' and 'hh:mm:ss' format
- will also recognize '.sss' milliseconds suffix, if any
- if L is left to default 0, it will be computed from StrLen(P)

function IsRawUTF8DynArray(typeinfo: pointer): boolean;

*Check if the TypeInfo() points to an "array of RawUTF8"
- e.g. returns true for TypeInfo(TRawUTF8DynArray) or other sub-types defined as "type aNewType = type TRawUTF8DynArray"

function IsRowID(FieldName: PUTF8Char): boolean; overload;

*Returns TRUE if the specified field name is either 'ID', either 'ROWID'*

function IsRowID(FieldName: PUTF8Char; FieldLen: integer): boolean; overload;

*Returns TRUE if the specified field name is either 'ID', either 'ROWID'*
function IsRowIDShort(const FieldName: shortstring): boolean; overload;
  Returns TRUE if the specified field name is either 'ID', either 'ROWID'

function isSelect(P: PUTF8Char; SelectClause: PRawUTF8=nil): boolean;
  Return true if the parameter is void or begin with a 'SELECT' SQL statement
  - used to avoid code injection and to check if the cache must be flushed
  - VACUUM, PRAGMA, or EXPLAIN statements also return true, since they won't change the data content
  - WITH recursive statement expect no INSERT/UPDATE/DELETE pattern in the SQL
  - if P^ is a SELECT and SelectClause is set to a variable, it would contain the field names, from SELECT ...field names... FROM

function IsString(P: PUTF8Char): boolean;
  Test if the supplied buffer is a "string" value or a numerical value (floating point or integer),
  according to the characters within
  - this version will recognize null/false/true as strings
  - e.g. IsString('0')=false, IsString('abc')=true, IsString('null')=true

function IsStringJSON(P: PUTF8Char): boolean;
  Test if the supplied buffer is a "string" value or a numerical value (floating or integer), according to
  the JSON encoding schema
  - this version will NOT recognize JSON null/false/true as strings
  - e.g. IsStringJSON('0')=false, IsStringJSON('abc')=true, but IsStringJSON('null')=false
  - will follow the JSON definition of number, i.e. '0123' is a string (i.e. '0' is excluded at the beginning
  of a number) and '123' is not a string

function IsUrlValid(P: PUTF8Char): boolean;
  Checks if the supplied UTF-8 text don't need URI encoding
  - returns TRUE if all its chars are non-void plain ASCII-7 RFC compatible identifiers (0..9a..zA..Z_.~)

function IsValidJSON(P: PUTF8Char; len: PtrInt): boolean; overload;
  Test if the supplied buffer is a correct JSON value

function IsValidJSON(const s: RawUTF8): boolean; overload;
  Test if the supplied buffer is a correct JSON value

function IsValidUTF8(const source: RawUTF8): Boolean; overload;
  Returns TRUE if the supplied buffer has valid UTF-8 encoding
  - will also refuse #0 characters within the buffer

function IsValidUTF8(source: PUTF8Char): Boolean; overload;
  Returns TRUE if the supplied buffer has valid UTF-8 encoding
  - will stop when the buffer contains #0

function IsValidUTF8(source: PUTF8Char; sourcelen: PtrInt): Boolean; overload;
  Returns TRUE if the supplied buffer has valid UTF-8 encoding
  - will also refuse #0 characters within the buffer

function IsValidUTF8WithoutControlChars(source: PUTF8Char): Boolean; overload;
  Returns TRUE if the supplied buffer has valid UTF-8 encoding with no #1..#31 control characters
  - supplied input is a pointer to a #0 ended text buffer
function IsValidUTF8WithoutControlChars(const source: RawUTF8): Boolean; overload;
Returns TRUE if the supplied buffer has valid UTF-8 encoding with no #0..#31 control characters
- supplied input is a RawUTF8 variable

function IsVoid(const text: RawUTF8): boolean;
Check all character within text are spaces or control chars
- i.e. a faster alternative to trim(text)="

function IsWinAnsi(WideText: PWideChar; Length: integer): boolean; overload;
Return TRUE if the supplied unicode buffer only contains WinAnsi characters
- i.e. if the text can be displayed using ANSI_CHARSET

function IsWinAnsi(WideText: PWideChar): boolean; overload;
Return TRUE if the supplied unicode buffer only contains WinAnsi characters
- i.e. if the text can be displayed using ANSI_CHARSET

function IsWinAnsiU(UTF8Text: PUTF8Char): boolean;
Return TRUE if the supplied UTF-8 buffer only contains WinAnsi characters
- i.e. if the text can be displayed using ANSI_CHARSET

function IsWinAnsiU8Bit(UTF8Text: PUTF8Char): boolean;
Return TRUE if the supplied UTF-8 buffer only contains WinAnsi 8 bit characters
- i.e. if the text can be displayed using ANSI_CHARSET with only 8 bit unicode characters (e.g. no "tm" or such)

function IsZero(const Values: TRawUTF8DynArray): boolean; overload;
Returns TRUE if Value is nil or all supplied Values[] equal "

function IsZero(const dig: THash128): boolean; overload;
Returns TRUE if all 16 bytes of this 128-bit buffer equal zero
- e.g. a MD5 digest, or an AES block

function IsZero(const dig: THash512): boolean; overload;
Returns TRUE if all 64 bytes of this 512-bit buffer equal zero
- e.g. a SHA-512 digest

function IsZero(const dig: THash160): boolean; overload;
Returns TRUE if all 20 bytes of this 160-bit buffer equal zero
- e.g. a SHA-1 digest

function IsZero(const dig: THash384): boolean; overload;
Returns TRUE if all 48 bytes of this 384-bit buffer equal zero
- e.g. a SHA-384 digest

function IsZero(P: pointer; Length: integer): boolean; overload;
Returns TRUE if all bytes equal zero

function IsZero(const dig: THash256): boolean; overload;
Returns TRUE if all 32 bytes of this 256-bit buffer equal zero
- e.g. a SHA-256 digest, or a TECCSignature result

function IsZero(const Values: TIntegerDynArray): boolean; overload;
Returns TRUE if Value is nil or all supplied Values[] equal 0
function IsZero(const Values: TInt64DynArray): boolean; overload;
  Returns TRUE if Value is nil or all supplied Values[] equal 0

function IsZeroSmall(P: pointer; Length: PtrInt): boolean;
  Returns TRUE if all of a few bytes equal zero
  - to be called instead of IsZero() e.g. for 1..8 bytes

function JSONArrayCount(P, PMax: PUTF8Char): integer; overload;
  Compute the number of elements of a JSON array
  - this will handle any kind of arrays, including those with nested JSON objects or arrays
  - incoming P^ should point to the first char after the initial '[' (which may be a closing ']')
  - this overloaded method will abort if P reaches a certain position: for really HUGE arrays, it is
    faster to allocate the content within the loop, not ahead of time

function JSONArrayCount(P: PUTF8Char): integer; overload;
  Compute the number of elements of a JSON array
  - this will handle any kind of arrays, including those with nested JSON objects or arrays
  - incoming P^ should point to the first char AFTER the initial '[' (which may be a closing ']')
  - returns -1 if the supplied input is invalid, or the number of identified items in the JSON array
    buffer

function JSONArrayDecode(P: PUTF8Char; out Values: TPUTF8CharDynArray): boolean;
  Retrieve all elements of a JSON array
  - this will handle any kind of arrays, including those with nested JSON objects or arrays
  - incoming P^ should point to the first char AFTER the initial '[' (which may be a closing ']')
  - returns false if the supplied input is invalid
  - returns true on success, with Values[] pointing to each unescaped value, may be a JSON string,
    object, array of constant

function JSONArrayItem(P: PUTF8Char; Index: integer): PUTF8Char;
  Go to the #nth item of a JSON array
  - implemented via a fast SAX-like approach: the input buffer is not changed, nor no memory buffer
    allocated neither content copied
  - returns nil if the supplied index is out of range
  - returns a pointer to the index-nth item in the JSON array (first index=0)
  - this will handle any kind of arrays, including those with nested JSON objects or arrays
  - incoming P^ should point to the first initial '[' char

procedure JSONBufferReformat(P: PUTF8Char; out result: RawUTF8; Format: TTextWriterJSONFormat=jsonHumanReadable);
  Formats and indents a JSON array or document to the specified layout
  - just a wrapper around TTextWriter.AddJSONReformat() method
  - WARNING: the JSON buffer is decoded in-place, so P^ WILL BE modified

function JSONBufferReformatToFile(P: PUTF8Char; const Dest: TFileName; Format: TTextWriterJSONFormat=jsonHumanReadable): boolean;
  Formats and indents a JSON array or document as a file
  - just a wrapper around TTextWriter.AddJSONReformat() method
  - WARNING: the JSON buffer is decoded in-place, so P^ WILL BE modified
procedure JSONBufferToXML(P: PUTF8Char; const Header, NameSpace: RawUTF8; out result: RawUTF8);

Convert a JSON array or document into a simple XML content
- just a wrapper around TTextWriter.AddJSONToXML, with an optional header before the XML
converted data (e.g. XMLUTF8_HEADER), and an optional name space content node which will
nest the generated XML data (e.g. '<contents xmlns="http://www.w3.org/2001/XMLSchema-instance">') - the corresponding ending token will be appended after (e.g. '</contents>')
- WARNING: the JSON buffer is decoded in-place, so P^ WILL BE modified

function JSONDecode(P: PUTF8Char; out Values: TNameValuePUTF8CharDynArray; HandleValuesAsObjectOrArray: Boolean=false): PUTF8Char; overload;

Decode the supplied UTF-8 JSON content into an array of name/value pairs
- this procedure will decode the JSON content in-memory, i.e. the PUTF8Char array is created
inside JSON, which is therefore modified: make a private copy first if you want to reuse the JSON
content
- the supplied JSON buffer should stay available until Name/Value pointers from returned Values[]
are accessed
- if HandleValuesAsObjectOrArray is TRUE, then this procedure will handle JSON arrays or objects
- support enhanced JSON syntax, e.g. '{name:"John",year:1972}' is decoded just like
'{"name":"John","year":1972}'

Used for DI-2.1.2 (page 2545).

function JSONDecode(P: PUTF8Char; const Names: array of RawUTF8; Values: PValuePUTF8CharArray; HandleValuesAsObjectOrArray: Boolean=false): PUTF8Char; overload;

Decode the supplied UTF-8 JSON content for the supplied names
- data will be set in Values, according to the Names supplied e.g.
  JSONDecode(P,['name','year'],Values) -> Values[0]^='John'; Values[1]^='1972';
- if any supplied name wasn't found its corresponding Values[] will be nil
- this procedure will decode the JSON content in-memory, i.e. the PUTF8Char array is created
inside P, which is therefore modified: make a private copy first if you want to reuse the JSON
content
- if HandleValuesAsObjectOrArray is TRUE, then this procedure will handle JSON arrays or objects
- if ValuesLen is set, ValuesLen[] will contain the length of each Values[]
- returns a pointer to the next content item in the JSON buffer

Used for DI-2.1.2 (page 2545).
procedure JSONDecode(var JSON: RawUTF8; const Names: array of RawUTF8; Values: PValuePUTF8CharArray; HandleValuesAsObjectOrArray: Boolean=false); overload;

 Decode the supplied UTF-8 JSON content for the supplied names
 - data will be set in Values, according to the Names supplied e.g.
   JSONDecode(JSON,['name','year'],@Values) -> Values[0].Value='John'; Values[1].Value='1972';
 - if any supplied name wasn't found its corresponding Values[] will be nil
 - this procedure will decode the JSON content in-memory, i.e. the Putf8Char array is created inside JSON, which is therefore modified: make a private copy first if you want to reuse the JSON content
 - if HandleValuesAsObjectOrArray is TRUE, then this procedure will handle JSON arrays or objects
 - support enhanced JSON syntax, e.g. '{name:"John",year:1972}' is decoded just like
   '"name":"John","year":1972"

 Used for DI-2.1.2 (page 2545).

procedure JSONDecode(var JSON: RawJSON; const Names: array of RawUTF8; Values: PValuePUTF8CharArray; HandleValuesAsObjectOrArray: Boolean=false); overload;

 Decode the supplied UTF-8 JSON content for the supplied names
 - an overloaded function when the JSON is supplied as a RawJSON variable

 Used for DI-2.1.2 (page 2545).

function JSONDecode(var JSON: RawUTF8; const aName: RawUTF8='result'; wasString: PBoolean=nil; HandleValuesAsObjectOrArray: Boolean=false): RawUTF8; overload;

 Decode the supplied UTF-8 JSON content for the one supplied name
 - this function will decode the JSON content in-memory, so willunescape it in-place: it must be called only once with the same JSON data

 Used for DI-2.1.2 (page 2545).

function JSONEncode(const NameValuePairs: array of const): RawUTF8; overload;

 Encode the supplied data as an UTF-8 valid JSON object content
 - data must be supplied two by two, as Name,Value pairs, e.g.
   JSONEncode(['name','John','year',1972]) = '"name":"John","year":1972"
 - or you can specify nested arrays or objects with '['..']' or '{'..'}':
   J := JSONEncode(['doc',"{"name","John","abc","["a","b","c"]"},"id",123]);
   assert(J='"doc":{"name":"John","abc":["a","b","c"],"id":123}');
 - note that, due to a Delphi compiler limitation, cardinal values should be type-casted to Int64() (otherwise the integer mapped value will be converted)
 - you can pass nil as parameter for a null JSON value

 Used for DI-2.1.2 (page 2545).
function JSONEncode(const Format: RawUTF8; const Args,Params: array of const): RawUTF8; overload;

Encode the supplied (extended) JSON content, with parameters, as an UTF-8 valid JSON object content
- in addition to the JSON RFC specification strict mode, this method will handle some BSON-like extensions, e.g. unquoted field names:
  aJSON := JSONEncode('{id:?,%:{name:?,birthyear:?}}',['doc'],[10,'John',1982]);
- you can use nested _Obj() / _Arr() instances
  aJSON := JSONEncode('{%:{$in:[?,?]}}','type','food','snack']);
aJSON := JSONEncode('{}:{$in:[]}','type:','food','snack']));
  // will both return
  '{"type":{"$in:['food','snack']}}'
- if the SynMongoDB unit is used in the application, the MongoDB Shell syntax will also be recognized to create TBSONVariant, like
  new Date()   ObjectI
  see @http://docs.mongodb.org/manual/reference/mongodb-extended-json
  aJSON := JSONEncode('{name:,field:%/i}'
  // will return
  '{"name":"John","field":{"$regex":"acme.*corp","$options":"i"}}'
- will call internally _JSONFastFmt() to create a temporary TDocVariant with all its features - so is slightly slower than other JSONEncode* functions

Used for DI-2.1.2 (page 2545).

function JSONEncodeArrayDouble(const Values: array of double): RawUTF8; overload;
Encode the supplied floating-point array data as a valid JSON array

function JSONEncodeArrayInteger(const Values: array of integer): RawUTF8; overload;
Encode the supplied integer array data as a valid JSON array

function JSONEncodeArrayOfConst(const Values: array of const; WithoutBraces: boolean=false): RawUTF8; overload;
Encode the supplied array data as a valid JSON array content
- if WithoutBraces is TRUE, no [] will be generated
- note that, due to a Delphi compiler limitation, cardinal values should be type-casted to Int64() (otherwise the integer mapped value will be converted)

procedure JSONEncodeArrayOfConst(const Values: array of const; WithoutBraces: boolean; var result: RawUTF8); overload;
Encode the supplied array data as a valid JSON array content
- if WithoutBraces is TRUE, no [] will be generated
- note that, due to a Delphi compiler limitation, cardinal values should be type-casted to Int64() (otherwise the integer mapped value will be converted)

function JSONEncodeArrayUTF8(const Values: array of RawUTF8): RawUTF8; overload;
Encode the supplied UTF8 array data as an UTF-8 valid JSON array content

procedure JSONencodeNameSQLValue(const Name,SQLValue: RawUTF8; var result: RawUTF8);
Encode as JSON {'name':value} object, from a potential SQL quoted value
- will unquote the SQLValue using TTextWriter.AddQuotedStringAsJSON()
function JSONObjectAsJSONArrays(JSON: PUTF8Char; out keys, values: RawUTF8): boolean;

Convert one JSON object into two JSON arrays of keys and values
- i.e. makes the following transformation:
  \{key1:val1,key2:val2,...\} \rightarrow \[key1,key2,...\] + \[val1,val2,...\]
- this function won't allocate any memory during its process, nor modify the JSON input buffer
- is the reverse of the TTextWriter.AddJSONArraysAsJSONObject() method

function JsonObjectByPath(JsonObject, PropPath: PUTF8Char): PUTF8Char;

Go to a property of a JSON object, by its full path, e.g. 'parent.child'
- implemented via a fast SAX-like approach: the input buffer is not changed, nor no memory buffer allocated neither content copied
- returns nil if the supplied property path does not exist
- returns a pointer to the matching item in the JSON object
- this will handle any kind of objects, including those with nested JSON objects or arrays
- incoming P^ should point to the first initial '{' char

function JsonObjectItem(P: PUTF8Char; const PropName: RawUTF8; PropNameFound: PRawUTF8=nil): PUTF8Char;

Go to a named property of a JSON object
- implemented via a fast SAX-like approach: the input buffer is not changed, nor no memory buffer allocated neither content copied
- returns nil if the supplied property name does not exist
- returns a pointer to the matching item in the JSON object
- this will handle any kind of objects, including those with nested JSON objects or arrays
- incoming P^ should point to the first initial '{' char

function JSONObjectPropCount(P: PUTF8Char): integer;

Compute the number of fields in a JSON object
- this will handle any kind of objects, including those with nested JSON objects or arrays
- incoming P^ should point to the first char after the initial '{' (which may be a closing '}')

function JsonObjectsByPath(JsonObject, PropPath: PUTF8Char): RawUTF8;

Return all matching properties of a JSON object
- here the PropPath could be a comma-separated list of full paths, e.g. 'Prop1,Prop2' or 'Obj1.Obj2.Prop1,Obj1.Prop2'
- returns '' if no property did match
- returns a JSON object of all matching properties
- this will handle any kind of objects, including those with nested JSON objects or arrays
- incoming P^ should point to the first initial '{' char

function JsonPropNameValid(P: PUTF8Char): boolean;

Returns TRUE if the given text buffer contains simple characters as recognized by JSON extended syntax
- follow GetJSONPropName and GotoNextJSONObjectOrArray expectations

function JSONReformat(const JSON: RawUTF8; Format: TTextWriterJSONFormat=jsonHumanReadable): RawUTF8;

Formats and indents a JSON array or document to the specified layout
- just a wrapper around TTextWriter.AddJSONReformat, making a private of the supplied JSON buffer (so that JSON content would stay untouched)
function JSONReformatToFile(const JSON: RawUTF8; const Dest: TFileName; Format: TTextWriterJSONFormat=jsonHumanReadable): boolean;
  Formats and indents a JSON array or document as a file
  - just a wrapper around TTextWriter.AddJSONReformat, making a private of the supplied JSON buffer (so that JSON content would stay untouched)

function JSONRetrieveStringField(P: PUTF8Char; out Field: PUTF8Char; out FieldLen: integer; ExpectNameField: boolean): PUTF8Char;
  Retrieve a pointer to JSON string field content
  - returns either ':' for name field, either '}',',' for value field
  - returns nil on JSON content error
  - this function won't touch the JSON buffer, so you can call it before using in-place escape process via JSONDecode() or GetJSONField()

function JSONToVariant(const JSON: RawUTF8; Options: TDocVariantOptions=[dvoReturnNullForUnknownProperty]; AllowDouble: boolean=false): variant;
  Retrieve a variant value from a JSON UTF-8 text as per RFC 8259, RFC 7159, RFC 7158
  - follows TTextWriter.AddVariant() format (calls GetVariantFromJSON)
  - will instantiate either an Integer, Int64, currency, double or string value (as RawUTF8), guessing the best numeric type according to the textual content, and string in all other cases, except TryCustomVariants points to some options (e.g. @JSON_OPTIONS[true] for fast instance) and input is a known object or array, either encoded as strict-JSOn (i.e. {...} or [...] or some extended (e.g. BSON) syntax
  - this overloaded procedure will make a temporary copy before JSON parsing and return the variant as result

function JSONToVariantDynArray(const JSON: RawUTF8): TVariantDynArray;
  Convert a JSON array into a dynamic array of variants
  - will use a TDocVariantData temporary storage

procedure JSONToVariantInPlace(var Value: Variant; JSON: PUTF8Char; Options: TDocVariantOptions=[dvoReturnNullForUnknownProperty]; AllowDouble: boolean=false);
  Retrieve a variant value from a JSON buffer as per RFC 8259, RFC 7159, RFC 7158
  - follows TTextWriter.AddVariant() format (calls GetVariantFromJSON)
  - will instantiate either an Integer, Int64, currency, double or string value (as RawUTF8), guessing the best numeric type according to the textual content, and string in all other cases, except TryCustomVariants points to some options (e.g. @JSON_OPTIONS[true] for fast instance) and input is a known object or array, either encoded as strict-JSOn (i.e. {...} or [...] or some extended (e.g. BSON) syntax
  - warning: the JSON buffer will be modified in-place during process - use a temporary copy or the overloaded functions with RawUTF8 parameter if you need to access it later

function JSONToXML(const JSON: RawUTF8; const Header: RawUTF8=XMLUTF8_HEADER; const NameSpace: RawUTF8=''): RawUTF8;
  Convert a JSON array or document into a simple XML content
  - just a wrapper around TTextWriter.AddJSONToXML, making a private copy of the supplied JSON buffer using TSynTempBuffer (so that JSON content would stay untouched)
  - the optional header is added at the beginning of the resulting string
  - an optional name space content node could be added around the generated XML, e.g. '<content>'
procedure KahanSum(const Data: double; var Sum, Carry: double);

Compute the sum of values, using a running compensation for lost low-order bits
- a naive "Sum := Sum + Data" will be restricted to 53 bits of resolution, so will eventually result in an incorrect number
- Kahan algorithm keeps track of the accumulated error in integer operations, to achieve a precision of more than 100 bits
- see https://en.wikipedia.org/wiki/Kahan_summation_algorithm

procedure KB(bytes: Int64; out result: TShort16; nospace: boolean); overload;

Convert a size to a human readable value power-of-two metric value
- append EB, PB, TB, GB, MB, KB or B symbol with or without preceding space
- for EB, PB, TB, GB, MB and KB, add one fractional digit

function KB(bytes: Int64; nospace: boolean): TShort16; overload;

Delphi 2007 is buggy as hell convert a size to a human readable value
- append EB, PB, TB, GB, MB, KB or B symbol with or without preceding space
- for EB, PB, TB, GB, MB and KB, add one fractional digit

function KB(const buffer: RawByteString): TShort16; overload;

Delphi 2007 is buggy as hell convert a string size to a human readable value
- append EB, PB, TB, GB, MB, KB or B symbol
- for EB, PB, TB, GB, MB and KB, add one fractional digit

function KB(bytes: Int64): TShort16; overload;

Convert a size to a human readable value
- append EB, PB, TB, GB, MB, KB or B symbol with preceding space
- for EB, PB, TB, GB, MB and KB, add one fractional digit

function KBNoSpace(bytes: Int64): TShort16;

Delphi 2007 is buggy as hell convert a size to a human readable value
- append EB, PB, TB, GB, MB, KB or B symbol without preceding space
- for EB, PB, TB, GB, MB and KB, add one fractional digit

procedure KBU(bytes: Int64; var result: RawUTF8);

Convert a size to a human readable value
- append EB, PB, TB, GB, MB, KB or B symbol
- for EB, PB, TB, GB, MB and KB, add one fractional digit

function kr32(crc: cardinal; buf: PAnsiChar; len: PtrInt): cardinal;

Standard Kernighan & Ritchie hash from "The C programming Language", 3rd edition
- simple and efficient code, but too much collisions for THasher
- kr32() is 898.8 MB/s - crc32cfast() 1.7 GB/s, crc32csse42() 4.3 GB/s

function LogEscape(source: PAnsiChar; sourcelen: integer; var temp: TLogEscape; enabled: boolean=true): PAnsiChar;

Fill TLogEscape stack buffer with the (hexadecimal) chars of the input binary
- up to LOGESCAPELEN (i.e. 200) bytes will be escaped and appended to a Local temp: TLogEscape variable, using the EscapeBuffer() low-level function
- you can then log the resulting escaped text by passing the returned PAnsiChar as % parameter to a TSynLog.Log() method
- the "enabled" parameter can be assigned from a process option, avoiding to process the escape if verbose logs are disabled
- used e.g. to implement logBinaryFrameContent option for WebSockets
function LogEscapeFull(source: PAnsiChar; sourcelen: integer): RawUTF8; overload;
  Returns a text buffer with the (hexadecimal) chars of the input binary
  - is much slower than LogEscape/EscapeToShort, but has no size limitation

function LogEscapeFull(const source: RawByteString): RawUTF8; overload;
  Returns a text buffer with the (hexadecimal) chars of the input binary
  - is much slower than LogEscape/EscapeToShort, but has no size limitation

procedure LogToTextFile(Msg: RawUTF8);
  Log a message to a local text file
  - the text file is located in the executable directory, and its name is simply the executable file
    name with the '.log' extension instead of '.exe'
  - format contains the current date and time, then the Msg on one line
    - date and time format used is 'YYYYMMDD hh:mm:ss (i.e. ISO-8601)'

function LowerCase(const S: RawUTF8): RawUTF8;
  Fast conversion of the supplied text into lowercase
  - this will only convert 'A'..'Z' into 'a'..'z' (no NormToLower use), and will therefore be correct with
    true UTF-8 content

procedure LowerCaseCopy(Text: PUTF8Char; Len: PtrInt; var result: RawUTF8);
  Fast conversion of the supplied text into lowercase
  - this will only convert 'A'..'Z' into 'a'..'z' (no NormToLower use), and will therefore be correct with
    true UTF-8 content

procedure LowerCaseSelf(var S: RawUTF8);
  Fast in-place conversion of the supplied variable text into lowercase
  - this will only convert 'A'..'Z' into 'a'..'z' (no NormToLower use), and will therefore be correct with
    true UTF-8 content, but only for 7 bit

function LowerCaseU(const S: RawUTF8): RawUTF8;
  Fast conversion of the supplied text into 8 bit lowercase
  - this will not only convert 'A'..'Z' into 'a'..'z', but also accentuated latin characters ('E' acute into 'e'
    e.g.), using NormToLower[] array
  - it will therefore decode the supplied UTF-8 content to handle more than 7 bit of ascii characters

function LowerCaseUnicode(const S: RawUTF8): RawUTF8;
  Accurate conversion of the supplied UTF-8 content into the corresponding lower-case Unicode
  characters
  - this version will use the Operating System API, and will therefore be much slower than
    LowerCase/LowerCaseU versions, but will handle all kind of unicode characters

function MaxInteger(const Values: TIntegerDynArray; ValuesCount: PtrInt; MaxStart: integer=-1): Integer;
  Find the maximum 32-bit integer in Values[]
function MedianQuickSelect(const OnCompare: TOnValueGreater; n: integer; var TempBuffer: TSynTempBuffer): integer;

*Compute the median of a serie of values, using "Quickselect"*
- based on the algorithm described in "Numerical recipes in C", Second Edition
- expect the values information to be available from a comparison callback
- this version will use a temporary index list to exchange items order (supplied as a TSynTempBuffer), so won't change the supplied values themself
- returns the index of the median Value

function MedianQuickSelectInteger(Values: PIntegerArray; n: integer): integer;

*Compute the median of an integer serie of values, using "Quickselect"*
- warning: the supplied Integer array is modified in-place during the process, and won't be fully sorted on output (this is no QuickSort alternative)

function MicroSecToString(Micro: QWord): TShort16; overload;

*Convert a micro seconds elapsed time into a human readable value*
- append 'us', 'ms', 's', 'm', 'h' and 'd' symbol for the given value range, with two fractional digits

procedure MicroSecToString(Micro: QWord; out result: TShort16); overload;

*Delphi 2007 is buggy as hell convert a micro seconds elapsed time into a human readable value*
- append 'us', 'ms', 's', 'm', 'h' and 'd' symbol for the given value range, with two fractional digits

procedure MoveSmall(Source, Dest: Pointer; Count:_PTRUInt);

*An alternative Move() function tuned for small unaligned counts*
- warning: expects Count>0 and Source/Dest not nil
- warning: doesn't support buffers overlapping

procedure mul64x64(const left, right: QWord; out product: THash128Rec);

*Fast computation of two 64-bit unsigned integers into a 128-bit value*
function MultiEventAdd(var EventList; const Event: TMethod): boolean;

Low-level wrapper to add a callback to a dynamic list of events
- by default, you can assign only one callback to an Event: but by storing it as a dynamic array of events, you can use this wrapper to add one callback to this list of events
- if the event was already registered, do nothing (i.e. won't call it twice)
- since this function uses an unsafe typeless EventList parameter, you should not use it in high-level code, but only as wrapper within dedicated methods
- will add Event to EventList[] unless Event is already registered
- is used e.g. by TTextWriter as such:
  ...
  fEchos: array of TOnTextWriterEcho;
  ...
  procedure EchoAdd(const aEcho: TOnTextWriterEcho);
  ...
  procedure TTextWriter.EchoAdd(const aEcho: TOnTextWriterEcho);
  begin
    MultiEventAdd(fEchos,TMethod(aEcho));
  end;
then callbacks are then executed as such:
  if fEchos<>nil then
    for i := 0 to length(fEchos)-1 do
      fEchos[i](self,fEchoBuf);
  - use MultiEventRemove() to un-register a callback from the list

function MultiEventFind(const EventList; const Event: TMethod): integer;

Low-level wrapper to check if a callback is in a dynamic list of events
- by default, you can assign only one callback to an Event: but by storing it as a dynamic array of events, you can use this wrapper to check if a callback has already been registered to this list of events
- used internally by MultiEventAdd() and MultiEventRemove() functions

procedure MultiEventMerge(var DestList; const ToBeAddedList);

Low-level wrapper to add one or several callbacks from another list of events
- all events of the ToBeAddedList would be added to DestList
- the list is not checked for duplicates

procedure MultiEventRemove(var EventList; Index: Integer); overload;

Low-level wrapper to remove a callback from a dynamic list of events
- same as the same overloaded procedure, but accepting an EventList[] index to identify the Event to be suppressed
procedure MultiEventRemove(var EventList; const Event: TMethod); overload;

**Low-level wrapper to remove a callback from a dynamic list of events**
- by default, you can assign only one callback to an Event: but by storing it as a dynamic array of events, you can use this wrapper to remove one callback already registered by MultiEventAdd() to this list of events
- since this function uses an unsafe typeless EventList parameter, you should not use it in high-level code, but only as wrapper within dedicated methods
- is used e.g. by TTextWriter as such:

```pascal
... fEchos: array of TOnTextWriterEcho;
...
procedure EchoRemove(const aEcho: TOnTextWriterEcho);
...
procedure TTextWriter.EchoRemove(const aEcho: TOnTextWriterEcho);
begin
  MultiEventRemove(fEchos, TMethod(aEcho));
end;
```

function MultiPartFormDataAddField(const FieldName, FieldValue: RawUTF8; var MultiPart: TMultiPartDynArray): boolean;

**Encode a field in a multipart array**
- FieldName: field name of the part
- FieldValue: value of the field
- MultiPart: where the part is added

function MultiPartFormDataAddFile(const FileName: TFileName; var MultiPart: TMultiPartDynArray; const Name: RawUTF8 = ''): boolean;

**Encode a file in a multipart array**
- FileName: file to encode
- MultiPart: where the part is added
- Name: name of the part, is empty the name 'File###' is generated

function MultiPartFormDataDecode(const MimeType, Body: RawUTF8; var MultiPart: TMultiPartDynArray): boolean;

**Decode multipart/form-data POST request content**
- following RFC1867

function MultiPartFormDataEncode(const MultiPart: TMultiPartDynArray; var MultiPartContentType, MultiPartContent: RawUTF8): boolean;

**Encode multipart fields and files**
- only one of them can be used because MultiPartFormDataDecode must implement both decodings
- MultiPart: parts to build the multipart content from, which may be created using MultiPartFormDataAddFile/MultiPartFormDataAddField
- MultiPartContentType: variable returning Content-Type: multipart/form-data; boundary=xxx
  where xxx is the first generated boundary
- MultiPartGenerated: generated multipart content

function NeedsJsonEscape(P: PUTF8Char; PLen: integer): boolean; overload;

**Returns TRUE if the given text buffers would be escaped when written as JSON**
- e.g. if contains " or \ characters, as defined by http://www.ietf.org/rfc/rfc4627.txt
function NeedsJsonEscape(const Text: RawUTF8): boolean; overload;
  Returns TRUE if the given text buffers would be escaped when written as JSON
  - e.g. if contains " or \ characters, as defined by http://www.ietf.org/rfc/rfc4627.txt

function NeedsJsonEscape(P: PUTF8Char): boolean; overload;
  Returns TRUE if the given text buffers would be escaped when written as JSON
  - e.g. if contains " or \ characters, as defined by http://www.ietf.org/rfc/rfc4627.txt

function NewSynLocker: PSynLocker;
  Allocate and initialize a TSynLocker instance
  - caller should call result^.DoneAndFreemem when not used any more

function NextGrow(capacity: integer): integer;
  Compute the new capacity when expanding an array of items
  - handle tiny, small, medium, large and huge sizes properly to reduce memory usage and
    maximize performance

function NextNotSpaceCharIs(var P: PUTF8Char; ch: AnsiChar): boolean;
  Check if the next character not in [#1..' ' ] matches a given value
  - first ignore any non space character
  - then returns TRUE if P^=ch, setting P to the character after ch
  - or returns FALSE if P^<>ch, leaving P at the level of the unexpected char

function NextUTF8UCS4(var P: PUTF8Char): cardinal;
  Get the UCS4 char stored in P^ (decode UTF-8 if necessary)

procedure NotifySortedIntegerChanges(old, new: PIntegerArray; oldn, newn: PtrInt;
  const added, deleted: TOnNotifySortedIntegerChange; const sender);
  Compares two 32-bit signed sorted integer arrays, and call event handlers to notify the
  corresponding modifications in an O(n) time
  - items in both old[] and new[] arrays are required to be sorted

function NowToString(Expanded: boolean=true; FirstTimeChar: AnsiChar=' '): RawUTF8;
  Retrieve the current Date, in the ISO 8601 layout, but expanded and ready to be displayed

function NowUTC: TDateTime;
  Returns the current UTC system date and time
  - SysUtils.Now returns local time: this function returns the system time expressed in Coordinated
    Universal Time (UTC)
  - under Windows, will use GetSystemTimeAsFileTime() so will achieve about 16 ms of resolution
  - under POSIX, will call clock_gettime(CLOCK_REALTIME_COARSE)

function NowUTCToString(Expanded: boolean=true; FirstTimeChar: AnsiChar=' '): RawUTF8;
  Retrieve the current UTC Date, in the ISO 8601 layout, but expanded and ready to be displayed
function ObjArrayAdd(var aObjArray; aItem: TObject): PtrInt;
  Wrapper to add an item to a T*ObjArray dynamic array storage
  - as expected by TJSONSerializer.RegisterTypeObjArrayForJSON()
  - could be used as such (note the T*ObjArray type naming convention):
    TUserObjArray = array of TUser;
    ...
    var arr: TUserObjArray;
    user: TUser;
    ...
    try
      user := TUser.Create;
      user.Name := 'Name';
      index := ObjArrayAdd(arr, user);
    ...
    finally
      ObjArrayClear(arr); // release all items
    end;

  - return the index of the item in the dynamic array

function ObjArrayAddCount(var aObjArray; aItem: TObject; var aObjArrayCount: integer): PtrInt;
  Wrapper to add an item to a T*ObjArray dynamic array storage
  - this overloaded function will use a separated variable to store the items count, so will be slightly
    faster: but you should call SetLength() when done, to have an array as expected by
    TJSONSerializer.RegisterTypeObjArrayForJSON()
  - return the index of the item in the dynamic array

function ObjArrayAddFrom(var aDestObjArray; const aSourceObjArray): PtrInt;
  Wrapper to add items to a T*ObjArray dynamic array storage
  - aSourceObjArray[] items are just copied to aDestObjArray, which remains untouched
  - return the new number of the items in aDestObjArray

procedure ObjArrayAddOnce(var aObjArray; aItem: TObject);
  Wrapper to add once an item to a T*ObjArray dynamic array storage
  - as expected by TJSONSerializer.RegisterTypeObjArrayForJSON()
  - if the object is already in the array (searching by address/reference, not by content), return its
    current index in the dynamic array
  - if the object does not appear in the array, add it at the end

function ObjArrayAddOnceFrom(var aDestObjArray; const aSourceObjArray): PtrInt;
  - aSourceObjArray[] items are just copied to aDestObjArray, which remains untouched
  - will first check if aSourceObjArray[] items are not already in aDestObjArray
  - return the new number of the items in aDestObjArray

function ObjArrayAppend(var aDestObjArray, aSourceObjArray): PtrInt;
  Wrapper to add and move items to a T*ObjArray dynamic array storage
  - aSourceObjArray[] items will be owned by aDestObjArray[], therefore aSourceObjArray is set to nil
  - return the new number of the items in aDestObjArray
procedure ObjArrayClear(var aObjArray); overload;
Wrapper to release all items stored in a T*ObjArray dynamic array
- as expected by TJSONSerializer.RegisterObjArrayForJSON()
- you should always use ObjArrayClear() before the array storage is released, e.g. in the owner class destructor
- will also set the dynamic array length to 0, so could be used to re-use an existing T*ObjArray

procedure ObjArrayClear(var aObjArray; aCount: integer); overload;
Wrapper to release all items stored in a T*ObjArray dynamic array
- this overloaded function will use the supplied array length as parameter
- you should always use ObjArrayClear() before the array storage is released, e.g. in the owner class destructor
- will also set the dynamic array length to 0, so could be used to re-use an existing T*ObjArray

procedure ObjArrayClear(var aObjArray; aContinueOnException: boolean; aCount: PInteger=nil); overload;
Wrapper to release all items stored in a T*ObjArray dynamic array
- as expected by TJSONSerializer.RegisterObjArrayForJSON()
- you should always use ObjArrayClear() before the array storage is released, e.g. in the owner class destructor
- will also set the dynamic array length to 0, so could be used to re-use an existing T*ObjArray

function ObjArrayCount(const aObjArray): integer;
Wrapper to count all not nil items in a T*ObjArray dynamic array storage
- as expected by TJSONSerializer.RegisterObjArrayForJSON()

function ObjArrayDelete(var aObjArray; aCount: integer; aItem: TObject): PInt;
Wrapper to delete an item in a T*ObjArray dynamic array storage
- search is performed by address/reference, not by content
- do nothing if the item is not found in the dynamic array

function ObjArrayDelete(var aObjArray; aItem: TObject): PInt; overload;
Wrapper to delete an item in a T*ObjArray dynamic array storage
- as expected by TJSONSerializer.RegisterObjArrayForJSON()
- search is performed by address/reference, not by content
- do nothing if the item is not found in the dynamic array

function ObjArrayDelete(var aObjArray; aItemIndex: PInt; aContinueOnException: boolean=false; aCount: PInteger=nil); overload;
Wrapper to delete an item in a T*ObjArray dynamic array storage
- as expected by TJSONSerializer.RegisterObjArrayForJSON()
- do nothing if the index is out of range in the dynamic array

procedure ObjArrayDelete(var aObjArray; aItemIndex: PInt; aContinueOnException: boolean=false; aCount: PInteger=nil); overload;
Wrapper to delete an item in a T*ObjArray dynamic array storage
- as expected by TJSONSerializer.RegisterObjArrayForJSON()
- do nothing if the index is out of range in the dynamic array

function ObjArrayFind(const aObjArray; aItem: TObject): PInt; overload;
Wrapper to search an item in a T*ObjArray dynamic array storage
- as expected by TJSONSerializer.RegisterObjArrayForJSON()
- search is performed by address/reference, not by content
- returns -1 if the item is not found in the dynamic array
function ObjArrayFind(const aObjArray; aCount: integer; aItem: TObject): PTrInt;
 overload;
 Wrapper to search an item in a T*ObjArray dynamic array storage
 - as expected by TJSONSerializer.RegisterObjArrayForJSON()
 - search is performed by address/reference, not by content
 - returns -1 if the item is not found in the dynamic array

procedure ObjArrayObjArrayClear(var aObjArray);
 Wrapper to release all items stored in an array of T*ObjArray dynamic array
 - e.g. aObjArray may be defined as "array of array of TSynFilter"

procedure ObjArraysClear(const aObjArray: array of pointer);
 Wrapper to release all items stored in several T*ObjArray dynamic arrays
 - as expected by TJSONSerializer.RegisterObjArrayForJSON()

procedure ObjArraySetLength(var aObjArray; aLength: integer);
 Wrapper to set the length of a T*ObjArray dynamic array storage
 - could be used as an alternative to SetLength() when you do not know the exact T*ObjArray type

procedure ObjArraySort(var aObjArray; Compare: TDynArraySortCompare);
 Wrapper to sort the items stored in a T*ObjArray dynamic array
 - as expected by TJSONSerializer.RegisterObjArrayForJSON()

function ObjArrayToJSON(const aObjArray; aOptions: TTextWriterWriteObjectOptions=[woDontStoreDefault]): RawUTF8;
 Wrapper to serialize a T*ObjArray dynamic array as JSON
 - as expected by TJSONSerializer.RegisterObjArrayForJSON()

function ObjectsToJSON(const Names: array of RawUTF8; const Values: array of TObject; Options: TTextWriterWriteObjectOptions=[woDontStoreDefault]): RawUTF8;
 Will serialize set of TObject into its UTF-8 JSON representation
 - follows ObjectToJSON()/TTextWriter.WriterObject() functions output
 - if Names is not supplied, the corresponding class names would be used

function ObjectToJSON(Value: TObject; Options: TTextWriterWriteObjectOptions=[woDontStoreDefault]): RawUTF8;
 Will serialize any TObject into its UTF-8 JSON representation
 - serialize as JSON the published integer, Int64, floating point values, TDateTime (stored as ISO 8601 text), string, variant and enumerate (e.g. boolean) properties of the object (and its parents)
 - would set twoForceJSONStandard to force standard (non-extended) JSON
 - the enumerates properties are stored with their integer index value
 - will write also the properties published in the parent classes
 - nested properties are serialized as nested JSON objects
 - any TCollection property will also be serialized as JSON arrays
 - you can add some custom serializers for ANY Delphi class, via mORMot.pas' TJSONSerializer.RegisterCustomSerializer() class method
 - call internally TJSONSerializer.WriteObject() method (or fallback to TJSONWriter if mORMot.pas is not linked to the executable)
procedure ObjectToVariant(Value: TObject; var result: variant; Options: TTextWriterWriteObjectOptions); overload;
    Will convert any TObject into a TDocVariant document instance
    - a faster alternative to _Json(ObjectToJSON(Value),Options)
    - note that the result variable should already be cleared: no VarClear() is done by this function
    - would be used e.g. by VarRecToVariant() function
    - if you expect lazy-loading of a TObject, see TObjectVariant.New()

function ObjectToVariant(Value: TObject; EnumSetsAsText: boolean=false): variant; overload;
    Will convert any TObject into a TDocVariant document instance
    - a faster alternative to _JsonFast(ObjectToJSON(Value))
    - if you expect lazy-loading of a TObject, see TObjectVariant.New()

procedure ObjectToVariant(Value: TObject; out Dest: variant); overload;
    Will convert any TObject into a TDocVariant document instance
    - a slightly faster alternative to Dest := _JsonFast(ObjectToJSON(Value))
    - this would convert the TObject by representation, using only serializable published properties:
      do not use this function to store temporary a class instance, but e.g. to store an object values in a
      NoSQL database
    - if you expect lazy-loading of a TObject, see TObjectVariant.New()

function OctToBin(const Oct: RawUTF8): RawByteString; overload;
    Conversion from octal C-like escape into binary data
    - \xxx is converted into a single xxx byte from octal, and \\ into \\

function OctToBin(Oct: PAnsiChar; Bin: PByte): PtrInt; overload;
    Conversion from octal C-like escape into binary data
    - \xxx is converted into a single xxx byte from octal, and \\ into \\
    - will stop the conversion when Oct^=#0 or when invalid \xxx is reached
    - returns the number of bytes written to Bin^

procedure OrMemory(Dest,Source: PByteArray; size: PtrInt);
    Logical OR of two memory buffers
    - will perform on all buffer bytes:
      Dest[i] := Dest[i] or Source[i];

procedure PatchCode(Old,New: pointer; Size: integer; Backup: pointer=nil;
    LeaveUnprotected: boolean=false);
    Self-modifying code - change some memory buffer in the code segment
    - if Backup is not nil, it should point to a Size array of bytes, ready to contain the overridden code
      buffer, for further hook disabling

procedure PatchCodePtrUInt(Code: PPtrUInt; Value: PPtrUInt; LeaveUnprotected:
    boolean=false);
    Self-modifying code - change one PtrUInt in the code segment
    Used for DI-2.1.4 (page 2547).

function Plural(const itemname: shortstring; itemcount: cardinal): shortstring;
    Write count number and append 's' (if needed) to form a plural English noun
    - for instance, Plural('row',100) returns '100 rows' with no heap allocation
procedure PointerToHex(aPointer: Pointer; var result: RawUTF8); overload;
  Fast conversion from a pointer data into hexa chars, ready to be displayed
  - use internally BinToHexDisplay()

function PointerToHex(aPointer: Pointer): RawUTF8; overload;
  Fast conversion from a pointer data into hexa chars, ready to be displayed
  - use internally BinToHexDisplay()

function PointerToHexShort(aPointer: Pointer): TShort16; overload;
  Fast conversion from a pointer data into hexa chars, ready to be displayed
  - use internally BinToHexDisplay()
  - such result type would avoid a string allocation on heap

function PosChar(Str: PUTF8Char; Chr: AnsiChar): PUTF8Char;
  Fast retrieve the position of a given character

function PosCharAny(Str: PUTF8Char; Characters: PAnsiChar): PUTF8Char;
  Fast retrieve the position of any value of a given set of characters
  - see also strspn() function which is likely to be faster

function PosEx(const SubStr, S: RawUTF8; Offset: PtrUInt=1): integer;
  Faster RawUTF8 Equivalent of standard StrUtils.PosEx

function PosExChar(Chr: AnsiChar; const Str: RawUTF8): PtrInt;
  Optimized version of PosEx() with search text as one AnsiChar

function PosI(uppersubstr: PUTF8Char; const str: RawUTF8): PtrInt;
  A non case-sensitive RawUTF8 version of Pos()
  - uppersubstr is expected to be already in upper case
  - this version handle only 7 bit ASCII (no accentuated characters)

function PosIU(substr: PUTF8Char; const str: RawUTF8): Integer;
  A non case-sensitive RawUTF8 version of Pos()
  - substr is expected to be already in upper case
  - this version will decode the UTF-8 content before using NormToUpper[]

function PropNamesValid(const Values: array of RawUTF8): boolean;
  Returns TRUE if the given text buffers contains A..Z,0..9,_ characters
  - use it with property names values (i.e. only including A..Z,0..9,_ chars)
  - this function won't check the first char the same way than PropNameValid()

function PropNameValid(P: PUTF8Char): boolean;
  Returns TRUE if the given text buffer contains a..z,A..Z,0..9,_ chars
  - should match most usual property names values or other identifier names in the business logic
  - i.e. can be tested via IdemPropName*() functions, and the MongoDB-like extended JSON syntax
  - as generated by dvoSerializeAsExtendedJson
  - first char must be alphabetical or '-', following chars can be alphanumerical or '_'

function PtrArrayAdd(var aPtrArray; aItem: pointer): integer;
  Wrapper to add an item to a array of pointer dynamic array storage

function PtrArrayAddOnce(var aPtrArray; aItem: pointer): integer;
  Wrapper to add once an item to a array of pointer dynamic array storage
procedure PtrArrayDelete(var aPtrArray; aIndex: integer; aCount: PInteger=nil); overload;
Wrapper to delete an item from a array of pointer dynamic array storage

function PtrArrayDelete(var aPtrArray; aItem: pointer; aCount: PInteger=nil): integer; overload;
Wrapper to delete an item from a array of pointer dynamic array storage

function PtrArrayFind(var aPtrArray; aItem: pointer): integer;
Wrapper to find an item to a array of pointer dynamic array storage

function PPtrUIntScan(P: PPtrUIntArray; Count: PtrInt; Value: PtrUInt): pointer;
Fast search of a pointer-sized unsigned integer in an pointer-sized integer array
- Count is the number of pointer-sized integer entries in P^'
- returns true if P^[index]=Value within Count entries
- returns false if Value was not found

function PPtrUIntScanExists(P: PPtrUIntArray; Count: PtrInt; Value: PtrUInt): boolean;
Fast search of a pointer-sized unsigned integer position in an pointer-sized integer array
- Count is the number of pointer-sized integer entries in P^'
- returns true if P^[index]=Value within Count entries
- returns false if Value was not found

function PPtrUIntScanIndex(P: PPtrUIntArray; Count: PtrInt; Value: PtrUInt): PtrInt;
Fast search of a pointer-sized unsigned integer position in an pointer-sized integer array
- Count is the number of pointer-sized integer entries in P^'
- return index of P^[index]=Value
- return -1 if Value was not found

procedure QuickSortCompare(const OnCompare: TOnValueGreater; Index: PIntegerArray; L,R: PtrInt);
Performs a QuickSort using a comparison callback

procedure QuickSortIndexedPUTF8Char(Values: PPUTF8CharArray; Count: Integer; var SortedIndexes: TCardinalDynArray; CaseSensitive: boolean=false);
Sort a dynamic array of PUTF8Char items, via an external array of indexes
- you can use FastFindIndexedPUTF8Char() for fast O(log(n)) binary search

procedure QuickSortInt64(ID,CoValues: PInt64Array; L, R: PtrInt); overload;
Sort a 64-bit Integer array, low values first

procedure QuickSortInt64(ID: PInt64Array; L, R: PtrInt); overload;
Sort a 64-bit signed Integer array, low values first

procedure QuickSortInteger(var ID: TIntegerDynArray); overload;
Sort an Integer array, low values first

procedure QuickSortInteger(ID: PIntegerArray; L, R: PtrInt); overload;
Sort an Integer array, low values first

procedure QuickSortInteger(ID,CoValues: PIntegerArray; L, R: PtrInt); overload;
Sort an Integer array, low values first

procedure QuickSortPointer(P: PPointerArray; L, R: PtrInt);
Sort a pointer array, low values first
procedure QuickSortPtrInt(P: PPtrIntArray; L, R: PtrInt);
Sort a PtrInt array, low values first

procedure QuickSortQWord(ID: PQWordArray; L, R: PtrInt); overload;
Sort a 64-bit unsigned Integer array, low values first
- QWord comparison are implemented correctly under FPC or Delphi 2009+ - older compilers will use fast and exact SortDynArrayQWord()

procedure QuickSortRawUTF8(var Values: TRawUTF8DynArray; ValuesCount: integer; CoValues: PIntegerDynArray=nil; Compare: TU8Compare=nil);
Sort a dynamic array of RawUTF8 items
- if CoValues is set, the integer items are also synchronized
- by default, exact (case-sensitive) match is used; you can specify a custom compare function if needed in Compare optional parameter

procedure QuickSortWord(ID: PWordArray; L, R: PtrInt);
Sort a 16 bit unsigned Integer array, low values first

procedure QuotedStr(const S: RawUTF8; Quote: AnsiChar; var result: RawUTF8); overload;
Format a text content with SQL-like quotes
- UTF-8 version of the function available in SysUtils
- this function implements what is specified in the official SQLite3 documentation: "A string constant is formed by enclosing the string in single quotes ('). A single quote within the string can be encoded by putting two single quotes in a row - as in Pascal."

function QuotedStr(const S: RawUTF8; Quote: AnsiChar=''''): RawUTF8; overload;
Format a text content with SQL-like quotes
- UTF-8 version of the function available in SysUtils
- this function implements what is specified in the official SQLite3 documentation: "A string constant is formed by enclosing the string in single quotes ('). A single quote within the string can be encoded by putting two single quotes in a row - as in Pascal."

function QuotedStrJSON(const aText: RawUTF8): RawUTF8; overload;
Convert UTF-8 content into a JSON string
- with proper escaping of the content, and surrounding " characters

procedure QuotedStrJSON(P: PUTF8Char; PLen: PtrInt; var result: RawUTF8; const aPrefix: RawUTF8=''; const aSuffix: RawUTF8=''); overload;
Convert UTF-8 buffer into a JSON string
- with proper escaping of the content, and surrounding " characters

procedure QuotedStrJSON(const aText: RawUTF8; var result: RawUTF8; const aPrefix: RawUTF8=''; const aSuffix: RawUTF8=''); overload;
Convert UTF-8 content into a JSON string
- with proper escaping of the content, and surrounding " characters

function QWordScanIndex(P: PQWordArray; Count: PtrInt; const Value: QWord): PtrInt;
Fast search of an integer position in an unsigned 64-bit integer array
- Count is the number of QWord entries in P^
function Random32(max: cardinal): cardinal; overload;
  Fast compute of some 32-bit random value, with a maximum (excluded) upper value
  - i.e. returns a value in range [0..max-1]
  - calls internally the overloaded Random32 function

function Random32: cardinal; overload;
  Fast compute of some 32-bit random value
  - will use (slow but) hardware-derivated RDRAND Intel x86/x64 opcode if available, or fast
gsl_rng_taus2 generator by Pierre L'Ecuyer (which period is 2^88, i.e. about 10^26) if the CPU
doesn't support it
  - will detect known AMD CPUs RDRAND bugs, and fallback to gsl_rng_taus2
  - consider Random32gsl to avoid slow RDRAND call (up to 1500 cycles needed!)
  - use rather TAESPRNG.Main.FillRandom() for cryptographic-level randomness
  - thread-safe function: each thread will maintain its own TLeecuyer table

function Random32gsl(max: cardinal): cardinal; overload;
  Fast compute of bounded 32-bit random value, using the gsl_rng_taus2 generator
  - calls internally the overloaded Random32gsl function

function Random32gsl: cardinal; overload;
  Fast compute of some 32-bit random value, using the gsl_rng_taus2 generator
  - Random32 may call RDRAND opcode on Intel CPUs, whereas this function will use well
documented, much faster, and proven Pierre L'Ecuyer software generator
  - may be used if you don't want/trust RDRAND, if you expect a well defined cross-platform
generator, or have higher performance expectations
  - use rather TAESPRNG.Main.FillRandom() for cryptographic-level randomness
  - thread-safe function: each thread will maintain its own TLeecuyer table

procedure Random32Seed(entropy: pointer=nil; entropylen: PtrInt=0);
  Seed the gsl_rng_taus2 Random32/Random32gsl generator
  - this seeding won't affect RDRAND Intel x86/x64 opcode generation
  - by default, gsl_rng_taus2 generator is re-seeded every 256KB, much more often than the Pierre
    L'Ecuyer's algorithm period of 2^88
  - you can specify some additional entropy buffer; note that calling this function with the same
    entropy again WON'T seed the generator with the same sequence (as with RTL's RandomSeed
    function), but initiate a new one
  - thread-specific function: each thread will maintain its own seed table

procedure RandomGUID(out result: TGUID); overload;
  Compute a random GUID value

function RandomGUID: TGUID; overload;
  Compute a random GUID value

function RawByteStringArrayConcat(const Values: array of RawByteString): RawByteString;
  Fast concatenation of several AnsiStrings

procedure RawByteStringToBytes(const buf: RawByteString; out bytes: TBytes);
  Creates a TBytes from a RawByteString memory buffer
function RawByteStringToStream(const aString: RawByteString): TStream;

Create a TStream from a string content
- uses RawByteString for byte storage, whatever the codepage is
- in fact, the returned TStream is a TRawByteString instance, since this function is just a wrapper around:
  result := TRawByteStringStream.Create(aString);

procedure RawByteStringToVariant(Data: PByte; DataLen: Integer; var Value: variant);

Create a TStream from a string content
- you can then use VariantToRawByteString() to retrieve the binary content

procedure RawByteStringToVariant(const Data: RawByteString; var Value: variant);

Create a RawByteString content into a variant varString
- you can then use VariantToRawByteString() to retrieve the binary content

procedure RawObjectsClear(o: PObject; n: integer);

Low-level function calling FreeAndNil(o^) successively n times

function RawUnicodeToString(const U: RawUnicode): string; overload;

Convert any Raw Unicode encoded string into a generic VCL Text
- uses StrLenW() and not length(U) to handle case when was used as buffer

function RawUnicodeToString(P: PWideChar; L: integer): string; overload;

Convert any Raw Unicode encoded buffer into a generic VCL Text

procedure RawUnicodeToString(P: PWideChar; L: integer; var result: string); overload;

Convert any Raw Unicode encoded buffer into a generic VCL Text

function RawUnicodeToSynUnicode(const Unicode: RawUnicode): SynUnicode; overload;

Convert any Raw Unicode encoded String into a generic SynUnicode Text

function RawUnicodeToSynUnicode(WideChar: PWideChar; WideCharCount: integer): SynUnicode; overload;

Convert any Raw Unicode encoded String into a generic SynUnicode Text

function RawUnicodeToUtf8(WideChar: PWideChar; WideCharCount: integer; Flags: TCharConversionFlags = [ccfNoTrailingZero]): RawUTF8; overload;

Convert a RawUnicode PWideChar into a UTF-8 string

procedure RawUnicodeToUtf8(WideChar: PWideChar; WideCharCount: integer; var result: RawUTF8; Flags: TCharConversionFlags = [ccfNoTrailingZero]); overload;

Convert a RawUnicode PWideChar into a UTF-8 string

function RawUnicodeToUtf8(Dest: PUTF8Char; DestLen: PtrInt; Source: PWideChar; SourceLen: PtrInt; Flags: TCharConversionFlags): PtrInt; overload;

Convert a RawUnicode UTF-16 PWideChar into a UTF-8 buffer
- replace system.UnicodeToUtf8 implementation, which is rather slow since Delphi 2009+
- append a trailing #0 to the ending PUTF8Char, unless ccfNoTrailingZero is set
- if ccfReplacementCharacterForUnmatchedSurrogate is set, this function will identify unmatched surrogate pairs and replace them with EF BF BD / FFFD   Unicode Replacement character - see https://en.wikipedia.org/wiki/Specials_(Unicode_block)
function RawUnicodeToUtf8(const Unicode: RawUnicode): RawUTF8; overload;
    Convert a RawUnicode string into a UTF-8 string

function RawUnicodeToUtf8(WideChar: PWideChar; WideCharCount: integer; out UTF8Length: integer): RawUTF8; overload;
    Convert a RawUnicode PWideChar into a UTF-8 string
    - this version doesn't resize the resulting RawUTF8 string, but return the new resulting RawUTF8 byte count into UTF8Length

function RawUnicodeToWinAnsi(const Unicode: RawUnicode): WinAnsiString; overload;
    Convert a RawUnicode string into a WinAnsi (code page 1252) string

function RawUnicodeToWinAnsi(WideChar: PWideChar; WideCharCount: integer): WinAnsiString; overload;
    Convert a RawUnicode PWideChar into a WinAnsi (code page 1252) string

procedure RawUnicodeToWinPChar(dest: PAnsiChar; source: PWideChar; WideCharCount: integer);
    Direct conversion of a Unicode encoded buffer into a WinAnsi PAnsiChar buffer

function RawUTF8ArrayToCSV(const Values: array of RawUTF8; const Sep: RawUTF8='',''): RawUTF8;
    Return the corresponding CSV text from a dynamic array of UTF-8 strings

function RawUTF8ArrayToQuotedCSV(const Values: array of RawUTF8; const Sep: RawUTF8='','; Quote: AnsiChar=''''): RawUTF8;
    Return the corresponding CSV quoted text from a dynamic array of UTF-8 strings
    - apply QuoteStr() function to each Values[] item

procedure RawUTF8DynArrayClear(var Value: TRawUTF8DynArray);
    Low-level finalization of a dynamic array of RawUTF8
    - faster than RTL Finalize() or setting nil

function RawUTF8DynArrayEquals(const A,B: TRawUTF8DynArray; Count: integer): boolean; overload;
    True if both TRawUTF8DynArray are the same for a given number of items
    - A and B are expected to have at least Count items
    - comparison is case-sensitive

function RawUTF8DynArrayLoadFromContains(Source: PAnsiChar; Value: PUTF8Char; ValueLen: PtrInt; CaseSensitive: boolean): PtrInt;
    Search in a RawUTF8 dynamic array BLOB content as stored by TDynArray.SaveTo
    - same as search within TDynArray.LoadFrom() with no memory allocation nor memory copy: so is much faster
    - will return -1 if no match or invalid data, or the matched entry index
**function** RawUTF8ToGUID(const text: RawByteString): TGUID;

*Convert some UTF-8 encoded text into a TGUID*
- expect e.g. '{3f2504e0-4f89-11d3-9a0c-0305e82c3301}' (with the {})
- return {00000000-0000-0000-0000-000000000000} if the supplied text buffer is not a valid TGUID

**procedure** RawUTF8ToVariant(const Txt: RawUTF8; var Value: TVarData; ExpectedValueType: cardinal); overload;

*Convert an UTF-8 encoded text buffer into a variant RawUTF8 varString*
- this overloaded version expects a destination variant type (e.g. varString varOleStr / varUString) - if the type is not handled, will raise an EVariantTypeCastError

**procedure** RawUTF8ToVariant(Txt: PUTF8Char; TxtLen: integer; var Value: variant); overload;

*Convert an UTF-8 encoded text buffer into a variant RawUTF8 varString*

**function** RawUTF8ToVariant(const Txt: RawUTF8): variant; overload;

*Convert an UTF-8 encoded string into a variant RawUTF8 varString*

**procedure** RCU(var src,dst; len: integer);

*Thread-safe move of a memory buffer using a simple Read-Copy-Update pattern*

**procedure** RCU128(var src,dst);

*Thread-safe move of a 128-bit value using a simple Read-Copy-Update pattern*

**procedure** RCU32(var src,dst);

*Thread-safe move of a 32-bit value using a simple Read-Copy-Update pattern*

**procedure** RCU64(var src,dst);

*Thread-safe move of a 64-bit value using a simple Read-Copy-Update pattern*

**procedure** RCUPtr(var src,dst);

*Thread-safe move of a pointer value using a simple Read-Copy-Update pattern*

**function** ReadStringFromStream(S: TStream; MaxAllowedSize: integer=255): RawUTF8;

*Read an UTF-8 text from a TStream*
- format is Length(Integer):Text, i.e. the one used by WriteStringToStream
- will return "" if there is no such text in the stream
- you can set a MaxAllowedSize value, if you know how long the size should be
- it will read from the current position in S: so if you just write into S, it could be a good idea to rewind it before call, e.g.: WriteStringToStream(Stream,aUTF8Text); Stream.Seek(0,soBeginning); str := ReadStringFromStream(Stream);

**procedure** RecordClear(var Dest; TypeInfo: pointer);

*Clear a record content*
- this unit includes a fast optimized asm version for x86 on Delphi
procedure RecordCopy(var Dest; const Source; TypeInfo: pointer);

Copy a record content from source to Dest
- this unit includes a fast optimized asm version for x86 on Delphi

function RecordEquals(const RecA, RecB; TypeInfo: pointer; PRecSize: PInteger=nil): boolean;

Check equality of two records by content
- will handle packed records, with binaries (byte, word, integer...) and string types properties
- will use binary-level comparison: it could fail to match two floating-point values because of rounding issues (Currency won't have this problem)

function RecordLoad(var Res; const Source: RawByteString; TypeInfo: pointer): boolean; overload;

Fill a record content from a memory buffer as saved by RecordSave()
- will use the Source length to detect and avoid any buffer overflow
- returns false if the Source buffer was incorrect, true on success

function RecordLoad(var Rec; Source: PAnsiChar; TypeInfo: pointer; Len: PInteger=nil; SourceMax: PAnsiChar=nil): PAnsiChar; overload;

Fill a record content from a memory buffer as saved by RecordSave()
- return nil if the Source buffer is incorrect
- in case of success, return the memory buffer pointer just after the read content, and set the Rec size, in bytes, into Len reference variable
- will use a proprietary binary format, with some variable-length encoding of the string length - note that if you change the type definition, any previously-serialized content will fail, maybe triggering unexpected GPF: you may use TypeInfoToHash() if you share this binary data across executables
- you can optionally provide in SourceMax the first byte after the input memory buffer, which will be used to avoid any unexpected buffer overflow - would be mandatory when decoding the content from any external process (e.g. a maybe-forged client) - only with slightly performance penalty

function RecordLoadBase64(Source: PAnsiChar; Len: Pointer; var Rec; TypeInfo: pointer; UriCompatible: boolean=false): boolean;

Read a record content from a Base-64 encoded content
- expects RecordSaveBase64() format, with a left-sided binary CRC

function RecordLoadJSON(var Rec; JSON: PUTF8Char;TypeInfo: pointer; EndOfObject: PUTF8Char=nil; CustomVariantOptions: PDocVariantOptions=nil): PUTF8Char; overload;

Fill a record content from a JSON serialization as saved by TTextWriter.AddRecordJSON / RecordSaveJSON
- will use default Base64 encoding over RecordSave() binary - or custom true JSON format (as set by TTextWriter.RegisterCustomJSONSerializer or via enhanced RTTI), if available
- returns nil on error, or the end of buffer on success
- warning: the JSON buffer will be modified in-place during process - use a temporary copy if you need to access it later or if the string comes from a constant (refcount=-1) - see e.g. the overloaded RecordLoadJSON()
**function** RecordLoadJSON(var Rec; const JSON: RawUTF8; TypeInfo: pointer; CustomVariantOptions: PDocVariantOptions=nil): boolean; overload;

*Fill a record content from a JSON serialization as saved by TTextWriter.AddRecordJSON / RecordSaveJSON*
- this overloaded function will make a private copy before parsing it, so is safe with a read/only or shared string - but slightly slower
- will use default Base64 encoding over RecordSave() binary - or custom true JSON format (as set by TTextWriter.RegisterCustomJSONSerializer or via enhanced RTTI), if available

**function** RecordSave(const Rec; Dest: PAnsiChar; TypeInfo: pointer): PAnsiChar; overload;

*Save a record content into a destination memory buffer*
- Dest must be at least RecordSaveLength() bytes long
- will handle packed records, with binaries (byte, word, integer...) and string types properties (but not with internal raw pointers, of course)
- will use a proprietary binary format, with some variable-length encoding of the string length - note that if you change the type definition, any previously-serialized content will fail, maybe triggering unexpected GPF: you may use TypeInfoToHash() if you share this binary data across executables
- warning: will encode generic string fields as AnsiString (one byte per char) prior to Delphi 2009, and as UnicodeString (two bytes per char) since Delphi 2009: if you want to use this function between UNICODE and NOT UNICODE versions of Delphi, you should use some explicit types like RawUTF8, WinAnsiString, SynUnicode or even RawUnicode/WideString

**procedure** RecordSave(const Rec; var Dest: TSynTempBuffer; TypeInfo: pointer); overload;

*Save a record content into a destination memory buffer*
- caller should make Dest.Done once finished with Dest.buf/Dest.len buffer

**function** RecordSave(const Rec; Dest: PAnsiChar; TypeInfo: pointer; out Len: integer): PAnsiChar; overload;

*Save a record content into a destination memory buffer*
- Dest must be at least RecordSaveLength() bytes long
- will return the Rec size, in bytes, into Len reference variable
- will handle packed records, with binaries (byte, word, integer...) and string types properties (but not with internal raw pointers, of course)
- will use a proprietary binary format, with some variable-length encoding of the string length - note that if you change the type definition, any previously-serialized content will fail, maybe triggering unexpected GPF: you may use TypeInfoToHash() if you share this binary data across executables
- warning: will encode generic string fields as AnsiString (one byte per char) prior to Delphi 2009, and as UnicodeString (two bytes per char) since Delphi 2009: if you want to use this function between UNICODE and NOT UNICODE versions of Delphi, you should use some explicit types like RawUTF8, WinAnsiString, SynUnicode or even RawUnicode/WideString
function RecordSave(const Rec; TypeInfo: pointer): RawByteString; overload;

Save a record content into a RawByteString
- will handle packed records, with binaries (byte, word, integer...) and string types properties (but not with internal raw pointers, of course)
- will use a proprietary binary format, with some variable-length encoding of the string length - note that if you change the type definition, any previously-serialized content will fail, maybe triggering unexpected GPF: you may use TypeInfoToHash() if you share this binary data across executables
- warning: will encode generic string fields as AnsiString (one byte per char) prior to Delphi 2009, and as UnicodeString (two bytes per char) since Delphi 2009: if you want to use this function between UNICODE and NOT UNICODE versions of Delphi, you should use some explicit types like RawUTF8, WinAnsiString, SynUnicode or even RawUnicode/WideString

function RecordSaveBase64(const Rec; TypeInfo: pointer; UriCompatible: boolean=false): RawUTF8;

Save a record content into a Base-64 encoded UTF-8 text content
- will use RecordSave() format, with a left-sided binary CRC32C

function RecordSaveBytes(const Rec; TypeInfo: pointer): TBytes;

Save a record content into a TBytes dynamic array
- could be used as an alternative to RawByteString's RecordSave()

function RecordSaveJSON(const Rec; TypeInfo: pointer; EnumSetsAsText: boolean=false): RawUTF8;

Save record into its JSON serialization as saved by TTextWriter.AddRecordJSON
- will use default Base64 encoding over RecordSave() binary - or custom true JSON format (as set by TTextWriter.RegisterCustomJSONSerializer or via enhanced RTTI), if available (following EnumSetsAsText optional parameter for nested enumerates and sets)

function RecordSaveLength(const Rec; TypeInfo: pointer; Len: PInteger=nil): integer;

Compute the number of bytes needed to save a record content using the RecordSave() function
- will return 0 in case of an invalid (not handled) record type (e.g. if it contains an unknown variant)
- optional Len parameter will contain the Rec memory buffer length, in bytes

function RecordTypeInfoSize(aRecordTypeInfo: pointer): integer;

Retrieve the record size from its low-level RTTI

procedure RecordZero(var Dest; TypeInfo: pointer);

Initialize a record content
- calls RecordClear() and FillCharFast() with 0
- do nothing if the TypeInfo is not from a record/object

procedure RedirectCode(Func, RedirectFunc: Pointer; Backup: PPatchCode=nil);

Self-modifying code - add an asm JUMP to a redirected function
- if Backup is not nil, it should point to a TPatchCode buffer, ready to contain the overridden code buffer, for further hook disabling

procedure RedirectCodeRestore(Func: pointer; const Backup: TPatchCode);

Self-modifying code - restore a code from its RedirectCode() backup
function RefCntDecFree(var refcnt: TRefCnt): boolean;

Low-level string/dynarray reference counter unprocess
- caller should have tested that refcnt>=0
- returns true if the managed variable should be released (i.e. refcnt was 1)
- on Delphi, RefCnt field is a 32-bit longint, whereas on FPC it is a SizeInt/PtrInt

function ReleaseInternalWindow(var aWindowName: string; var aWindow: HWND): boolean;

Delete the window resources used to receive Windows Messages
- must be called for each CreateInternalWindow() function
- both parameter values are then reset to "/0

procedure RemoveCommentsFromJSON(P: PUTF8Char);

Remove comments and trailing commas from a text buffer before passing it to JSON parser
- handle two types of comments: starting from // till end of line or /* ..... */ blocks anywhere in the text content
- trailing commas is replaced by ",", so resulting JSON is valid for parsers which not allows trailing commas (browsers for example)
- may be used to prepare configuration files before loading; for example we store server configuration in file config.json and put some comments in this file then code for loading is:

```pascal
var cfg: RawUTF8;

cfg := StringFromFile(ExtractFilePath(paramstr(0))+'Config.json');
RemoveCommentsFromJSON(@cfg[1]);
pLastChar := JSONToObject(sc,pointer(cfg),configValid);
```

function RenameInCSV(const OldValue, NewValue: RawUTF8; var CSV: RawUTF8; const Sep: RawUTF8 = ',', '): boolean;

Change a Value within a CSV string

procedure ReplaceSection(var Content: RawUTF8; const SectionName, NewSectionContent: RawUTF8); overload;

Replace a whole [Section] content by a new content
- create a new [Section] if none was existing

procedure ReplaceSection(SectionFirstLine: PUTF8Char; var Content: RawUTF8; const NewSectionContent: RawUTF8); overload;

Replace a whole [Section] content by a new content
- create a new [Section] if none was existing
- SectionFirstLine may have been obtained by FindSectionFirstLine() function above

procedure ResourceSynLZToRawByteStr(const ResName: string; out buf: RawByteString; Instance: THandle=0);

Creates a RawByteString memory buffer from an SynLZ-compressed embedded resource
- returns "" if the resource is not found
- this method would use SynLZDecompress() after ResourceToRawByteString(), with a ResType=PChar(10) (i.e. RC_DATA)
- you can specify a library (dll) resource instance handle, if needed
procedure ResourceToRawByteString(const ResName: string; ResType: PChar; out buf: RawByteString; Instance: THandle=0);

  Creates a RawByteString memory buffer from an embedded resource
  - returns "" if the resource is not found
  - warning: resources size may be rounded up to alignment
  - you can specify a library (dll) resource instance handle, if needed

procedure Reverse(const Values: TIntegerDynArray; ValuesCount: PtrInt; Reversed: PIntegerArray);

  Fill already allocated Reversed[] so that Reversed[Values[i]]=i

function SameTextU(const S1, S2: RawUTF8): Boolean;

  SameText() overloaded function with proper UTF-8 decoding
  - fast version using NormToUpper[] array for all Win-Ansi characters
  - this version will decode each UTF-8 glyph before using NormToUpper[]
  - current implementation handles UTF-16 surrogates as UTF8IComp()

function SameValue(const A, B: Double; DoublePrec: double = DOUBLE_SAME): Boolean;

  Compare to floating point values, with IEEE 754 double precision
  - use this function instead of raw = operator
  - the precision is calculated from the A and B value range
  - faster equivalent than SameValue() in Math unit
  - if you know the precision range of A and B, it's faster to check abs(A-B)<range

function SameValueFloat(const A, B: TSynExtended; DoublePrec: TSynExtended = DOUBLE_SAME): Boolean;

  Compare to floating point values, with IEEE 754 double precision
  - use this function instead of raw = operator
  - the precision is calculated from the A and B value range
  - faster equivalent than SameValue() in Math unit
  - if you know the precision range of A and B, it's faster to check abs(A-B)<range

procedure SaveJSON(const Value; TypeInfo: pointer; Options: TTextWriterOptions; var result: RawUTF8); overload;

  Serialize most kind of content as JSON, using its RTTI
  - is just a wrapper around TTextWriter.AddTypedJSON()
  - so would handle tkClass, tkEnumeration, tkSet, tkRecord, tkDynArray, tkVariant kind of content
    - other kinds would return 'null'
  - you can override serialization options if needed

function SaveJSON(const Value; TypeInfo: pointer; EnumSetsAsText: boolean=false): RawUTF8; overload;

  Serialize most kind of content as JSON, using its RTTI
  - is just a wrapper around TTextWriter.AddTypedJSON()
  - so would handle tkClass, tkEnumeration, tkSet, tkRecord, tkDynArray, tkVariant kind of content
    - other kinds would return 'null'

function ScanUTF8(P: PUTF8Char; PLen: PtrInt; const fmt: RawUTF8; const values: array of pointer; ident: PRawUTF8DynArray): integer; overload;

  Read text from P/PLen and store it into values[] according to fmt specifiers
function ScanUTF8(const text, fmt: RawUTF8; const values: array of pointer; ident: PRawUTF8DynArray=nil): integer; overload;
  Read and store text into values[] according to fmt specifiers
  - %d as PInteger, %D as PInt64, %u as PCardinal, %U as PQWord, %f as PDouble, %F as PCurrency,
    %x as 8 hexa chars to PInteger, %X as 16 hexa chars to PInt64, %s as PShortString (UTF-8
    encoded), %S as PRawUTF8, %L as PRawUTF8 (getting all text until the end of the line)
  - optionally, specifiers and any whitespace separated identifiers may be extracted and stored into
    the ident[] array, e.g. '%dFirstInt %s %DOneInt64' will store ['dFirstInt', 's', 'DOneInt64'] into ident

function SearchRecToDateTime(const F: TSearchRec): TDateTime;
Get a file date and time, from a FindFirst/FindNext search
- the returned timestamp is in local time, not UTC
- this method would use the F.Timestamp field available since Delphi XE2

function SearchRecValidFile(const F: TSearchRec): boolean;
Check if a FindFirst/FindNext found instance is actually a file

function SearchRecValidFolder(const F: TSearchRec): boolean;
Check if a FindFirst/FindNext found instance is actually a folder

function SetAppUserModelID(const AppUserModelID: string): boolean;
Under Windows 7 and later, will set an unique application-defined Application User Model ID
(AppUserModelID) that identifies the current process to the taskbar
- this identifier allows an application to group its associated processes and windows under a single
  taskbar button
- value can have no more than 128 characters, cannot contain spaces, and each section should be
  camel-cased, as such:
  CompanyName.ProductName.SubProduct.VersionInformation
Company and ProductName should always be used, while the SubProduct and
VersionInformation portions are optional and depend on the application's requirements
- if the supplied text does not contain an ',', 'ID.ID' will be used

procedure SetBit(var Bits; aIndex: PtrInt);
Set a particular bit into a bit array
- this function can't be inlined, whereas SetBitPtr() function can

procedure SetBit64(var Bits: Int64; aIndex: PtrInt);
Set a particular bit into a 64-bit integer bits (max aIndex is 63)

procedure SetBitCSV(var Bits; BitsCount: integer; var P: PUTF8Char);
Retrieve the next CSV separated bit index
- each bit was stored as BitIndex+1, i.e. 0 to mark end of CSV chunk
- several bits set to one can be regrouped via 'first-last,' syntax

procedure SetBitPtr(Bits: pointer; aIndex: PtrInt);
Set a particular bit into a bit array
- SetBit() can't be inlined, whereas this pointer-oriented function can

procedure SetCurrentThreadName(const Format: RawUTF8; const Args: array of const);
Name the current thread so that it would be easily identified in the IDE debugger
procedure SetExecutableVersion(const aVersionText: RawUTF8); overload;
Initialize ExeVersion global variable, supplying the version as text
- e.g. SetExecutableVersion('7.1.2.512');

procedure SetExecutableVersion(aMajor,aMinor,aRelease,aBuild: integer); overload;
Initialize ExeVersion global variable, supplying a custom version number
- by default, the version numbers will be retrieved at startup from the executable itself (if it was included at build time)
- but you can use this function to set any custom version numbers

procedure SetInt64(P: PUTF8Char; var result: Int64);
Get the 64-bit signed integer value stored in P^

procedure SetQWord(P: PUTF8Char; var result: QWord);
Get the 64-bit unsigned integer value stored in P^

procedure SetThreadName(ThreadID: TThreadID; const Format: RawUTF8; const Args: array of const);
Name a thread so that it would be easily identified in the IDE debugger
- you can force this function to do nothing by setting the NOSETTHREADNAME conditional, if you have issues with this feature when debugging your app
- most meaningful less characters (like 'TSQL') are trimmed to reduce the resulting length - which is convenient e.g. with POSIX truncation to 16 chars

procedure SetThreadNameDefault(ThreadID: TThreadID; const Name: RawUTF8);
Could be used to override SetThreadNameInternal()
- under Linux/FPC, calls pthread_setname_np API which truncates to 16 chars

procedure SetVariantByRef(const Source: Variant; var Dest: Variant);
Same as Dest := Source, but copying by reference
- i.e. VType is defined as varVariant or varByRef
- for instance, it will be used for late binding of TDocVariant properties, to let following statements work as expected:
  V := _Json('{arr:[1,2]}');
  V.arr.Add(3); // will work, since V.arr will be returned by reference
  writeln(V); // will write '{"arr":[1,2,3]}'

procedure SetVariantByValue(const Source: Variant; var Dest: Variant);
Same as Dest := Source, but copying by value
- will unreference any varByRef content
- will convert any string value into RawUTF8 (varString) for consistency

procedure SetVariantNull(var Value: variant);
Same as Value := Null, but slightly faster

function SetVariantUnRefSimpleValue(const Source: variant; var Dest: TVarData):
  boolean;
  Same as Dest := TVarData(Source) for simple values
  - will return TRUE for all simple values after varByRef unreference, and copying the unreferenced Source value into Dest raw storage
  - will return FALSE for not varByRef values, or complex values (e.g. string)
procedure ShortStringToAnsi7String(const source: shortstring; var result: RawUTF8);
overload;

Direct conversion of an ANSI-7 shortstring into an AnsiString
- can be used e.g. for names retrieved from RTTI to convert them into RawUTF8

function ShortStringToAnsi7String(const source: shortstring): RawByteString;
overload;

Direct conversion of an ANSI-7 shortstring into an AnsiString
- can be used e.g. for names retrieved from RTTI to convert them into RawUTF8

function ShortStringToUTF8(const source: ShortString): RawUTF8;

Direct conversion of a WinAnsi shortstring into a UTF-8 text
- call internally WinAnsiConvert fast conversion class

function SimpleDynArrayLoadFrom(Source: PAnsiChar; aTypeInfo: pointer; var Count, ElemSize: integer; NoHash32Check: boolean=false): pointer;
Wrap a simple dynamic array BLOB content as stored by TDynArray.SaveTo
- a "simple" dynamic array contains data with no reference count, e.g. byte, word, integer, cardinal, Int64, double or Currency
- same as TDynArray.LoadFrom() with no memory allocation nor memory copy: so is much faster than creating a temporary dynamic array to load the data
- will return nil if no or invalid data, or a pointer to the data array otherwise, with the items number stored in Count and the individual element size in ElemSize (e.g. 2 for a TWordDynArray)

function SimpleRoundTo2Digits(Value: Currency): Currency;
Simple, no banker rounding of a Currency value to only 2 digits
- #.##51 will round to #.##0.01 and #.##50 will be truncated to #.##
- implementation will use fast Int64 math to avoid any precision loss due to temporary floating-point conversion

procedure SimpleRoundTo2DigitsCurr64(var Value: Int64);
Simple, no banker rounding of a Currency value, stored as Int64, to only 2 digits
- #.##51 will round to #.##0.01 and #.##50 will be truncated to #.##
- implementation will use fast Int64 math to avoid any precision loss due to temporary floating-point conversion

procedure SleepHiRes(ms: cardinal);
Similar to Windows sleep() API call, to be truly cross-platform
- it should have a millisecond resolution, and handle ms=0 as a switch to another pending thread, i.e. under Windows will call SwitchToThread API

function SortDynArray128(const A,B): integer;
Compare two "array of THash128" elements

function SortDynArray256(const A,B): integer;
Compare two "array of THash256" elements

function SortDynArray512(const A,B): integer;
Compare two "array of THash512" elements

function SortDynArrayAnsiString(const A,B): integer;
Compare two "array of AnsiString" elements, with case sensitivity
function SortDynArrayAnsiStringI(const A,B): integer;
    Compare two "array of AnsiString" elements, with no case sensitivity

function SortDynArrayBoolean(const A,B): integer;
    Compare two "array of boolean" elements

function SortDynArrayByte(const A,B): integer;
    Compare two "array of byte" elements

function SortDynArrayCardinal(const A,B): integer;
    Compare two "array of cardinal" elements

function SortDynArrayDouble(const A,B): integer;
    Compare two "array of double" elements

function SortDynArrayFileName(const A,B): integer;
    Compare two "array of TFileName" elements, as file names
    - i.e. with no case sensitivity, and grouped by file extension
    - the expected string type is the generic RTL string, i.e. TFileName
    - calls internally GetFileNameWithoutExt() and AnsiCompareFileName()

function SortDynArrayInt64(const A,B): integer;
    Compare two "array of Int64" or "array of Currency" elements

function SortDynArrayInteger(const A,B): integer;
    Compare two "array of integer" elements

function SortDynArrayPointer(const A,B): integer;
    Compare two "array of TObject/pointer" elements

function SortDynArrayPUTF8Char(const A,B): integer;
    Compare two "array of PUTF8Char/PAnsiChar" elements, with case sensitivity

function SortDynArrayPUTF8CharI(const A,B): integer;
    Compare two "array of PUTF8Char/PAnsiChar" elements, with no case sensitivity

function SortDynArrayQWord(const A,B): integer;
    Compare two "array of QWord" elements
    - note that QWord(A)>QWord(B) is wrong on older versions of Delphi, so you should better use this function or CompareQWord() to properly compare two QWord values over CPUX86

function SortDynArrayRawByteString(const A,B): integer;
    Compare two "array of RawByteString" elements, with case sensitivity
    - can't use StrComp() or similar functions since RawByteString may contain #0

function SortDynArrayShortint(const A,B): integer;
    Compare two "array of shortint" elements

function SortDynArraySingle(const A,B): integer;
    Compare two "array of single" elements

function SortDynArraySmallint(const A,B): integer;
    Compare two "array of smallint" elements
function SortDynArrayString(const A, B): integer;
    Compare two "array of generic string" elements, with case sensitivity
    - the expected string type is the generic VCL string

function SortDynArrayStringI(const A, B): integer;
    Compare two "array of generic string" elements, with no case sensitivity
    - the expected string type is the generic VCL string

function SortDynArrayUnicodeString(const A, B): integer;
    Compare two "array of WideString/UnicodeString" elements, with case sensitivity

function SortDynArrayUnicodeStringI(const A, B): integer;
    Compare two "array of WideString/UnicodeString" elements, with no case sensitivity

function SortDynArrayVariant(const A, B): integer;
    Compare two "array of variant" elements, with case sensitivity

function SortDynArrayVariantComp(const A, B: TVarData; caseInsensitive: boolean): integer;
    Compare two "array of variant" elements, with or without case sensitivity
    - this low-level function is called by SortDynArrayVariant/VariantCompare
    - more optimized than the RTL function if A and B share the same type

function SortDynArrayVariantI(const A, B): integer;
    Compare two "array of variant" elements, with no case sensitivity

function SortDynArrayWord(const A, B): integer;
    Compare two "array of word" elements

function Split(const Str: RawUTF8; const SepStr: array of RawUTF8; const DestPtr: array of PRawUTF8): PtrInt; overload;
    Split a RawUTF8 string into several strings, according to SepStr separator
    - this overloaded function will fill a DestPtr[] array of PRawUTF8
    - if any DestPtr[]=nil, the item will be skipped
    - if input Str end before al SepStr[]=found, DestPtr[]=set to "
    - returns the number of values extracted into DestPtr[]

procedure Split(const Str, SepStr: RawUTF8; var LeftStr, RightStr: RawUTF8; ToUpperCase: boolean=false); overload;
    Split a RawUTF8 string into two strings, according to SepStr separator
    - if SepStr is not found, LeftStr=Str and RightStr="
    - if ToUpperCase is TRUE, then LeftStr and RightStr will be made uppercase

function Split(const Str, SepStr: RawUTF8; var LeftStr: RawUTF8; ToUpperCase: boolean=false): RawUTF8; overload;
    Returns the left part of a RawUTF8 string, according to SepStr separator
    - if SepStr is not found, returns Str
    - if SepStr is found, returns Str first chars until (and excluding) SepStr
function SplitRight(const Str: RawUTF8; SepChar: AnsiChar; LeftStr: PRawUTF8=nil): RawUTF8;

Returns the last occurrence of the given SepChar separated context
- e.g. SplitRight('01/2/34','/')='34'
- if SepChar doesn't appear, will return Str, e.g. SplitRight('123','/')='123'
- if LeftStr is supplied, the RawUTF8 it points to will be filled with the left part just before SepChar
" if SepChar doesn't appear

function SplitRights(const Str, SepChar: RawUTF8): RawUTF8;

Returns the last occurrence of the given SepChar separated context
- e.g. SplitRight('path/one\two/file.ext','/\')='file.ext', i.e. SepChars='/\' will be like ExtractFileName() over RawUTF8 string
- if SepChar doesn't appear, will return Str, e.g. SplitRight('123','/')='123'

procedure SQLAddWhereAnd(var where: RawUTF8; const condition: RawUTF8);

Add a condition to a SQL WHERE clause, with an ' and ' if where is not void

function SQLBegin(P: PUTF8Char): PUTF8Char;

Go to the beginning of the SQL statement, ignoring all blanks and comments
- used to check the SQL statement command (e.g. is it a SELECT?)

function StrCompFast(Str1, Str2: pointer): PtrInt;

Buffer-safe version of StrComp(), to be used with PUTF8Char/PAnsiChar
- pure pascal StrComp() won't access the memory beyond the string, but this function is defined for compatibility with SSE 4.2 expectations

function StrCompIL(P1,P2: PUTF8Char; L: Integer; Default: Integer=0): PtrInt;

Use our fast version of StrCompIL(), to be used with PUTF8Char

function StrCompL(P1,P2: PUTF8Char; L, Default: Integer): PtrInt;

Use our fast version of StrCompL(), to be used with PUTF8Char

function StrCompW(Str1, Str2: PWideChar): PtrInt;

Use our fast version of StrComp(), to be used with PWideChar

function strcspnpas(s,reject: pointer): integer;

Pure pascal version of strcspn(), to be used with PUTF8Char/PAnsiChar
- please note that this optimized version may read up to 3 bytes beyond reject but never after s end, so is safe e.g. over memory mapped files

function strcspnsse42(s,reject: pointer): integer;

SSE 4.2 version of strcspn(), to be used with PUTF8Char/PAnsiChar
- please note that this optimized version may read up to 15 bytes beyond the string; this is rarely a problem but it may generate protection violations, which could trigger fatal SIGABRT or SIGSEGV on Posix system
- could be used instead of strcspn() when you are confident about your s/reject input buffers, checking if cfSSE42 in CpuFeatures
**function StrCurr64(P: PAnsiChar; const Value: Int64): PAnsiChar;**

*Internal fast INTEGER Curr64 (value*10000) value to text conversion*
  - expect the last available temporary char position in P
  - return the last written char position (write in reverse order in P^)
  - will return 0 for Value=0, or a string representation with always 4 decimals (e.g. 1->'0.0001'
  500->'0.0500' 25000->'2.5000' 30000->'3.0000')
  - is called by Curr64ToPChar() and Curr64ToStr() functions

**function StreamSynLZ(Source: TCustomMemoryStream; Dest: TStream; Magic: cardinal): integer; overload;**

*Compress a data content using the SynLZ algorithm from one stream into another*
  - returns the number of bytes written to Dest
  - you should specify a Magic number to be used to identify the block

**function StreamSynLZ(Source: TCustomMemoryStream; const DestFile: TFileName; Magic: cardinal): integer; overload;**

*Compress a data content using the SynLZ algorithm from one stream into a file*
  - returns the number of bytes written to the destination file
  - you should specify a Magic number to be used to identify the block

**function StreamSynLZComputeLen(P: PAnsiChar; Len, aMagic: cardinal): integer;**

*Compute the real length of a given StreamSynLZ-compressed buffer*
  - allows to replace an existing appended content, for instance

**function StreamToRawByteString(aStream: TStream): RawByteString;**

*Read a TStream content into a String*
  - it will read binary or text content from the current position until the end (using TStream.Size)
  - uses RawByteString for byte storage, whatever the codepage is

**function StreamUnSynLZ(const Source: TFileName; Magic: cardinal): TMemoryStream; overload;**

*Uncompress using the SynLZ algorithm from one file into another*
  - returns a newly create memory stream containing the uncompressed data
  - returns nil if source file is invalid (e.g. invalid name or invalid content)
  - you should specify a Magic number to be used to identify the block
  - this function will also recognize the block at the end of the source file (if was appended to an
  existing data - e.g. a .mab at the end of a .exe)

**function StreamUnSynLZ(Source: TStream; Magic: cardinal): TMemoryStream; overload;**

*Uncompress using the SynLZ algorithm from one stream into another*
  - returns a newly create memory stream containing the uncompressed data
  - returns nil if source data is invalid
  - you should specify a Magic number to be used to identify the block
  - this function will also recognize the block at the end of the source stream (if was appended to an
  existing data - e.g. a .mab at the end of a .exe)
  - on success, Source will point after all read data (so that you can e.g. append several data blocks
to the same stream)

**function StrIComp(Str1, Str2: pointer): PtrInt;**

*Use our fast version of StrIComp(), to be used with PUTF8Char/PAnsiChar*
procedure StringBufferToUtf8(Source: PChar; out result: RawUTF8); overload;

Convert any generic VCL 0-terminated Text buffer into an UTF-8 string
- it will work as is with Delphi 2009+ (direct unicode conversion)
- under older version of Delphi (no unicode), it will use the current RTL codepage, as with
  WideString conversion (but without slow WideString usage)

function StringBufferToUtf8(Dest: PUTF8Char; Source: PChar; SourceChars: IntPtr): PUTF8Char; overload;

Convert any generic VCL Text buffer into an UTF-8 encoded buffer
- Dest must be able to receive at least SourceChars*3 bytes
- it will work as is with Delphi 2009+ (direct unicode conversion)
- under older version of Delphi (no unicode), it will use the current RTL codepage, as with
  WideString conversion (but without slow WideString usage)

procedure StringDynArrayToRawUTF8DynArray(const Source: TStringDynArray; var Result: TRawUTF8DynArray);

Convert the string dynamic array into a dynamic array of UTF-8 strings

function StringFromFile(const FileName: TFileName; HasNoSize: boolean=false): RawByteString;

Read a File content into a String
- content can be binary or text
- returns "" if file was not found or any read error occured
- will use GetFileSize() API by default, unless HasNoSize is defined, and read will be done using a
  buffer (required e.g. for char files under Linux)
- uses RawByteString for byte storage, whatever the codepage is

procedure StringListToRawUTF8DynArray(Source: TStringList; var Result: TRawUTF8DynArray);

Convert the string list into a dynamic array of UTF-8 strings

function StringReplaceAll(const S: RawUTF8; const OldNewPatternPairs: array of RawUTF8): RawUTF8; overload;

Fast version of several cascaded StringReplaceAll()

function StringReplaceAll(const S, OldPattern, NewPattern: RawUTF8): RawUTF8; overload;

Fast version of StringReplace(S, OldPattern, NewPattern,[rfReplaceAll]);

function StringReplaceAllProcess(const S, OldPattern, NewPattern: RawUTF8; found: integer): RawUTF8;

Actual replacement function called by StringReplaceAll() on first match
- not to be called as such, but defined globally for proper inlining

function StringReplaceChars(const Source: RawUTF8; OldChar, NewChar: AnsiChar): RawUTF8;

Fast replace of a specified char by a given string

function StringReplaceTabs(const Source,TabText: RawUTF8): RawUTF8;

Fast replace of all #9 chars by a given string

function StringToAnsi7(const Text: string): RawByteString;

Convert any generic VCL Text into Ansi 7 bit encoded String
- the Text content must contain only 7 bit pure ASCII characters
function StringToGUID(const text: string): TGUID;

- Convert some text into a TGUID
- expect e.g. '{3F2504E0-4F89-11D3-9A0C-0305E82C3301}' (with the {})
- return {00000000-0000-0000-0000-000000000000} if the supplied text buffer is not a valid TGUID

function StringToRawUnicode(const S: string): RawUnicode; overload;

- Convert any generic VCL Text into a Raw Unicode encoded String
- it's preferred to use TLanguageFile.StringToUTF8() method in mORMoti18n, which will handle full i18n of your application
- it will work as is with Delphi 2009+ (direct unicode conversion)
- under older version of Delphi (no unicode), it will use the current RTL codepage, as with WideString conversion (but without slow WideString usage)

function StringToRawUnicode(P: PChar; L: integer): RawUnicode; overload;

- Convert any generic VCL Text into a Raw Unicode encoded String
- it's preferred to use TLanguageFile.StringToUTF8() method in mORMoti18n, which will handle full i18n of your application
- it will work as is with Delphi 2009+ (direct unicode conversion)
- under older version of Delphi (no unicode), it will use the current RTL codepage, as with WideString conversion (but without slow WideString usage)

function StringToSynUnicode(const S: string): SynUnicode; overload;

- Convert any generic VCL Text into a SynUnicode encoded String
- it's preferred to use TLanguageFile.StringToUTF8() method in mORMoti18n, which will handle full i18n of your application
- it will work as is with Delphi 2009+ (direct unicode conversion)
- under older version of Delphi (no unicode), it will use the current RTL codepage, as with WideString conversion (but without slow WideString usage)

procedure StringToSynUnicode(const S: string; var result: SynUnicode); overload;

- Convert any generic VCL Text into a SynUnicode encoded String
- overloaded to avoid a copy to a temporary result string of a function

procedure StringToUTF8(Text: PChar; TextLen:_PTRInt; var result: RawUTF8); overload;

- Convert any generic VCL Text buffer into an UTF-8 encoded String
- it will work as is with Delphi 2009+ (direct unicode conversion)
- under older version of Delphi (no unicode), it will use the current RTL codepage, as with WideString conversion (but without slow WideString usage)

procedure StringToUTF8(const Text: string; var result: RawUTF8); overload;

- Convert any generic VCL Text into an UTF-8 encoded String
- this overloaded function use a faster by-reference parameter for the result

function StringToUTF8(const Text: string): RawUTF8; overload;

- Convert any generic VCL Text into an UTF-8 encoded String
- in the VCL context, it's preferred to use TLanguageFile.StringToUTF8() method from mORMoti18n, which will handle full i18n of your application
- it will work as is with Delphi 2009+ (direct unicode conversion)
- under older version of Delphi (no unicode), it will use the current RTL codepage, as with WideString conversion (but without slow WideString usage)
### Function `StringToWinAnsi` (const `Text: string`): `WinAnsiString`

Convert any generic VCL Text into WinAnsi (Win-1252) 8 bit encoded String

### Function `StrInt32` (P: `PAnsiChar`; `val: PTrInt`): `PAnsiChar`

*Internal fast integer val to text conversion*
- expect the last available temporary char position in P
- return the last written char position (write in reverse order in P^)
- typical use:
  ```pascal
  function Int32ToUTF8(Value: PTrInt): RawUTF8;
  var tmp: array[0..23] of AnsiChar;
  P: PAnsiChar;
  begin
    P := StrInt32(@tmp[23],Value);
    String(result,P,@tmp[23]-P);
  end;
  ```
- convert the input value as PTrInt, so as Int64 on 64-bit CPUs
- not to be called directly: use IntToStr() or Int32ToUTF8() instead

### Function `StrInt64` (P: `PAnsiChar`; `const val: Int64`): `PAnsiChar`

*Internal fast Int64 val to text conversion*
- same calling convention as with StrInt32() above

### Function `StrLenPas` (S: `pointer`): `PtrInt`

*Slower version of StrLen(), but which will never read beyond the string*
- this version won't access the memory beyond the string, so may be preferred to StrLen(), when using e.g. memory mapped files or any memory protected buffer

### Function `StrLenW` (S: `PWideChar`): `PtrInt`

*Our fast version of StrLen(), to be used with PWideChar*

### Function `StrPosI` (uppersubstr, str: `PUTF8Char`): `PUTF8Char`

*A non case-sensitive version of Pos()*
- uppersubstr is expected to be already in upper case
- this version handle only 7 bit ASCII (no accentuated characters)

### Function `strspnpas` (s, accept: `pointer`): `integer`

*Pure pascal version of strspn(), to be used with PUTF8Char/PAnsiChar*
- please note that this optimized version may read up to 3 bytes beyond accept but never after s end, so is safe e.g. over memory mapped files

### Function `strspnsse42` (s, accept: `pointer`): `integer`

*SSE 4.2 version of strspn(), to be used with PUTF8Char/PAnsiChar*
- please note that this optimized version may read up to 15 bytes beyond the string; this is rarely a problem but it may generate protection violations, which could trigger fatal SIGABRT or SIGSEGV on Posix system
- could be used instead of strspn() when you are confident about your s/accept input buffers, checking if cfSSE42 in CpuFeatures

### Function `StrToCurr64` (P: `PUTF8Char`; NoDecimal: `PBoolean=nil`): `Int64`

*Convert a string into its INTEGER Curr64 (value*10000) representation*
- this type is compatible with Delphi currency memory map with PInt64(@Curr)^
- fast conversion, using only integer operations
- if NoDecimal is defined, will be set to TRUE if there is no decimal, AND the returned value will be an Int64 (not a PInt64(@Curr)^)
function StrToCurrency(P: PUTF8Char): currency;
    Convert a string into its currency representation
    - will call StrToCurren64()

function StrUInt32(P: PAnsiChar; val: PtrUInt): PAnsiChar;
    Internal fast unsigned integer val to text conversion
    - expect the last available temporary char position in P
    - return the last written char position (write in reverse order in P^)
    - convert the input value as PtrUInt, so as QWord on 64-bit CPUs

function StrUInt64(P: PAnsiChar; const val: QWord): PAnsiChar;
    Internal fast unsigned Int64 val to text conversion
    - same calling convention as with StrInt32() above

function SumInteger(const Values: TIntegerDynArray; ValuesCount: PtrInt): Integer;
    Sum all 32-bit integers in Values[]

procedure SymmetricEncrypt(key: cardinal; var data: RawByteString);
    Naive symmetric encryption scheme using a 32-bit key
    - fast, but not very secure, since uses crc32ctab[] content as master cypher key: consider using
      SynCrypto proven AES-based algorithms instead

function SynchFolders(const Reference, Dest: TFileName; SubFolder: boolean=false;
    ByContent: boolean=false; WriteFileNameToConsole: boolean=false): integer;
    Ensure all files in Dest folder(s) do match the one in Reference
    - won't copy all files from Reference folders, but only update files already existing in Dest, which
      did change since last synchronization
    - will also process recursively nested folders if SubFolder is true
    - will use file content instead of file date check if ByContent is true
    - can optionally write the synched file name to the console
    - returns the number of files copied during the process

procedure SynLZCompress(P: PAnsiChar; PLen: integer; out Result: RawByteString;
    CompressionSizeTrigger: integer=100; CheckMagicForCompressed: boolean=false);
    Deprecated function - please call AlgoSynLZ.Compress() method

function SynLZCompress(P: PAnsiChar; PLen, DestLen: integer;
    CompressionSizeTrigger: integer=100; CheckMagicForCompressed: boolean=false): integer;
    Deprecated function - please call AlgoSynLZ.Compress() method

function SynLZCompress(const Data: RawByteString; CompressionSizeTrigger:
    integer=100; CheckMagicForCompressed: boolean=false): RawByteString; overload;
    Deprecated function - please call AlgoSynLZ.Compress() method

function SynLZCompressBytes(P: PAnsiChar; PLen: integer; CompressionSizeTrigger:
    integer=100): TByteDynArray; overload;
    Deprecated function - please call AlgoSynLZ.CompressBytes() method

function SynLZCompressBytes(Data: RawByteString; CompressionSizeTrigger:
    integer=100): TByteDynArray; overload;
    Deprecated function - please call AlgoSynLZ.DecompressBytes() method
procedure SynLZDecompress(P: PAnsiChar; PLen: integer; out Result: RawByteString; SafeDecompression: boolean=false); overload;

  Deprecated function - please call AlgoSynLZ.Decompress() method

function SynLZDecompress(P: PAnsiChar; PLen: integer; out Len: integer; var tmp: RawByteString): pointer; overload;

  Deprecated function - please call AlgoSynLZ.Decompress() method

function SynLZDecompress(const Data: RawByteString): RawByteString; overload;

  Deprecated function - please call AlgoSynLZ.Decompress() method

function SynLZDecompress(const Data: TByteDynArray): RawByteString; overload;

  Deprecated function - please call AlgoSynLZ.Decompress() method

function SynLZDecompress(const Data: RawByteString; out Len: integer; var tmp: RawByteString): pointer; overload;

  Deprecated function - please call AlgoSynLZ.Decompress() method

function SynLZDecompressBody(P,Body: PAnsiChar; PLen,BodyLen: integer; SafeDecompression: boolean=false): boolean;

  Deprecated function - please call AlgoSynLZ.DecompressBody() method

function SynLZDecompressHeader(P: PAnsiChar; PLen: integer): integer;

  Deprecated function - please call AlgoSynLZ.DecompressHeader() method

function SynLZDecompressPartial(P,Partial: PAnsiChar; PLen,PartialLen: integer): integer;

  Deprecated function - please call AlgoSynLZ.DecompressPartial() method

function SynRegisterCustomVariantType(aClass: TSynInvokeableVariantTypeClass): TSynInvokeableVariantType;

  Register a custom variant type to handle properties
  - this will implement an internal mechanism used to bypass the default _DispInvoke() implementation in Variant.pas, to use a faster version
  - is called in case of TSynTableVariant, TDocVariant, TBSONVariant or TSQLDBRowVariant

function SynUnicodeToString(const U: SynUnicode): string;

  Convert any SynUnicode encoded string into a generic VCL Text

function SynUnicodeToUtf8(const Unicode: SynUnicode): RawUTF8;

  Convert a SynUnicode string into a UTF-8 string

function TemporaryFileName: TFileName;

  Compute an unique temporary file name
  - following 'exename_01234567.tmp' pattern, in the system temporary folder

function TextToGUID(P: PUTF8Char; guid: PByteArray): PUTF8Char;

  Convert some text into its TGUID binary value
  - expect e.g. '3F2504E0-4F89-11D3-9AOC-0305EB82C3301' (without any {})
  - return nil if the supplied text buffer is not a valid TGUID
  - this will be the format used for JSON encoding, e.g.
    { "UID": "C9A846D3-9C61-4CB7-BFCD-EE2522C8F633" }
procedure TextToVariant(const aValue: RawUTF8; AllowVarDouble: boolean; out aDest: variant);
   Convert an UTF-8 encoded text buffer into a variant number or RawUTF8 varString
   - first try with GetNumericVariantFromJSON(), then fallback to RawUTF8ToVariant

function TextToVariantNumberType(JSON: PUTF8Char): cardinal;
  Identify either varInt64, varDouble, varCurrency types following JSON format
  - any non valid number is returned as varString
  - is used e.g. by GetVariantFromJSON() to guess the destination variant type
  - warning: supplied JSON is expected to be not nil

function TextToVariantNumberTypeNoDouble(JSON: PUTF8Char): cardinal;
  Identify either varInt64 or varCurrency types following JSON format
  - this version won't return varDouble, i.e. won't handle more than 4 exact decimals (as varCurrency), nor scientific notation with exponent (1.314e10)
  - this will ensure that any incoming JSON will converted back with its exact textual representation, without digit truncation due to limited precision
  - any non valid number is returned as varString
  - is used e.g. by GetVariantFromJSON() to guess the destination variant type
  - warning: supplied JSON is expected to be not nil

function TimeLogFromDateTime(const DateTime: TDateTime): TTimeLog;
  Get TTimeLog value from a given Delphi date and time
  - handle TTimeLog bit-encoded Int64 format
  - just a wrapper around PTimeLogBits(@aTime)^.From()
  - we defined such a function since TTimeLogBits(aTimeLog).From() won't change the aTimeLog variable content

function TimeLogFromFile(const FileName: TFileName): TTimeLog;
  Get TTimeLog value from a file date and time
  - handle TTimeLog bit-encoded Int64 format

function TimeLogFromUnixTime(const UnixTime: TUnixTime): TTimeLog;
  Get TTimeLog value from a given Unix seconds since epoch timestamp
  - handle TTimeLog bit-encoded Int64 format
  - just a wrapper around PTimeLogBits(@aTime)^.FromUnixTime()

function TimeLogNow: TTimeLog;
  Get TTimeLog value from current local system date and time
  - handle TTimeLog bit-encoded Int64 format

function TimeLogNowUTC: TTimeLog;
  Get TTimeLog value from current UTC system Date and Time
  - handle TTimeLog bit-encoded Int64 format

function TimeLogToDateTime(const Timestamp: TTimeLog): TDateTime;
  Date/Time conversion from a TTimeLog value
  - handle TTimeLog bit-encoded Int64 format
  - just a wrapper around PTimeLogBits(@Timestamp)^.ToDateTime
  - we defined such a function since TTimeLogBits(aTimeLog).ToDateTime gives an internall compiler error on some Delphi IDE versions (e.g. Delphi 6)
function TimeLogToUnixTime(const Timestamp: TTimeLog): TUnixTime;

Unix seconds since epoch timestamp conversion from a TTimeLog value
- handle TTimeLog bit-encoded Int64 format
- just a wrapper around PTimeLogBits(@Timestamp).ToUnixTime

function TimeToIso8601(Time: TDateTime; Expanded: boolean; FirstChar: AnsiChar='T'; WithMS: boolean=false): RawUTF8;

Basic Time conversion into ISO-8601
- use 'Thhmmss' format if not Expanded
- use 'Thh:mm:ss' format if Expanded
- if WithMS is TRUE, will append '.sss' for milliseconds resolution

function TimeToIso8601PChar(Time: TDateTime; P: PUTF8Char; Expanded: boolean; FirstChar: AnsiChar='T'; WithMS: boolean=false): PUTF8Char; overload;

Write a Time to P^ Ansi buffer
- if Expanded is false, 'Thhmmss' time format is used
- if Expanded is true, 'Thh:mm:ss' time format is used
- you can custom the first char in from of the resulting text time
- if WithMS is TRUE, will append 'sss' for milliseconds resolution

function TimeToIso8601PChar(P: PUTF8Char; Expanded: boolean; H,M,S,MS: PtrUInt; FirstChar: AnsiChar='T'; WithMS: boolean=false): PUTF8Char; overload;

Write a Time to P^ Ansi buffer
- if Expanded is false, 'Thhmmss' time format is used
- if Expanded is true, 'Thh:mm:ss' time format is used
- you can custom the first char in from of the resulting text time
- if WithMS is TRUE, will append MS as '.sss' for milliseconds resolution

function TimeToString: RawUTF8;

Retrieve the current Time (without Date), in the ISO 8601 layout
- useful for direct on screen logging e.g.

function TInt64DynArrayFrom(const Values: TIntegerDynArray): TInt64DynArray;

Quick helper to initialize a dynamic array of 64-bit integers from 32-bit values
- see also FromI64() for 64-bit signed integer values input

function TIntegerDynArrayFrom(const Values: array of integer): TIntegerDynArray;

Quick helper to initialize a dynamic array of integer from some constants
- can be used e.g. as:
  MyArray := TIntegerDynArrayFrom([1,2,3]);
- see also FromI32()

function TInt64DynArrayFrom64(const Values: TInt64DynArray; raiseExceptionOnOverflow: boolean=true): TIntegerDynArray;

Quick helper to initialize a dynamic array of integer from 64-bit integers
- will raise a ESynException if any Value[] can not fit into 32-bit, unless raiseExceptionOnOverflow is FALSE and the returned array slot is filled with maxInt/minInt

function ToCardinal(const text: RawUTF8; out value: cardinal; minimal: cardinal=0): boolean;

Get the unsigned 32-bit cardinal value stored in a RawUTF8 string
- returns TRUE if the supplied text was successfully converted into a cardinal
function ToDouble(const text: RawUTF8; out value: double): boolean;
Get a 64-bit floating-point value stored in a RawUTF8 string
- returns TRUE if the supplied text was successfully converted into a double

function ToInt64(const text: RawUTF8; out value: Int64): boolean;
Get the signed 64-bit integer value stored in a RawUTF8 string
- returns TRUE if the supplied text was successfully converted into an Int64

function ToInteger(const text: RawUTF8; out value: integer): boolean;
Get the signed 32-bit integer value stored in a RawUTF8 string
- returns TRUE if the supplied text was successfully converted into an integer

procedure ToText(C: TClass; var result: RawUTF8); overload;
Just a wrapper around vmtClassName to avoid a string/RawUTF8 conversion

function ToText(kind: TDocVariantKind): PShortString; overload;
Retrieve the text representation of a TDocVairnatKind

function ToText(C: TClass): RawUTF8; overload;
Just a wrapper around vmtClassName to avoid a string/RawUTF8 conversion

function ToText(const aIntelCPUFeatures: TIntelCpuFeatures; const Sep: RawUTF8=',:');
RawUTF8; overload;
Convert Intel CPU features as plain CSV text

function ToUTF8(Value: Int64): RawUTF8; overload;
Use our fast RawUTF8 version of IntToStr()

function ToUTF8(const V: Variant): RawUTF8; overload;
Convert any Variant into UTF-8 encoded String
- use VariantSaveJSON() instead if you need a conversion to JSON with custom parameters
- note: null will be returned as 'null'

function ToUTF8(const Text: string): RawUTF8; overload;
Convert any generic VCL Text into an UTF-8 encoded String

function ToUTF8(Value: PInt): RawUTF8; overload;
Use our fast RawUTF8 version of IntToStr()

function ToUTF8(const guid: TGUID): RawUTF8; overload;
Convert a TGUID into UTF-8 encoded text
- will return e.g. '3F2504E0-4F89-11D3-9A0C-0305E82C3301' (without the {})
- if you need the embracing { }, use GUIDToRawUTF8() function instead

function ToUTF8(const Ansi7Text: ShortString): RawUTF8; overload;
Convert any UTF-8 encoded shortstring Text into an UTF-8 encoded String
- expects the supplied content to be already ASCII-7 or UTF-8 encoded, e.g. a RTTI type or property name: it won't work with Ansi-encoded strings

function ToVarInt32(Value: PInt; Dest: PByte): PByte;
Convert an integer into a 32-bit variable-length integer buffer
- store negative values as cardinal two-complement, i.e. 0=0,1=1,2=-1,3=2,4=-2...

function ToVarInt64(Value: Int64; Dest: PByte): PByte;
Convert a Int64 into a 64-bit variable-length integer buffer
function ToVarString(const Value: RawUTF8; Dest: PByte): PByte;
    Convert a RawUTF8 into an UTF-8 encoded variable-length buffer

function ToVarUInt32(Value: cardinal; Dest: PByte): PByte;
    Convert a cardinal into a 32-bit variable-length integer buffer

function ToVarUInt32Length(Value:_PTR_UINT):_PTR_UINT;
    Return the number of bytes necessary to store a 32-bit variable-length integer
    - i.e. the ToVarUInt32() buffer size

function ToVarUInt32LengthWithData(Value:_PTR_UINT):_PTR_UINT;
    Return the number of bytes necessary to store some data with a its 32-bit variable-length integer length

function ToVarUInt64(Value: QWord; Dest: PByte): PByte;
    Convert a UInt64 into a 64-bit variable-length integer buffer

function TQWordDynArrayFrom(const Values: TCardinalDynArray): TQWordDynArray;
    Quick helper to initialize a dynamic array of 64-bit integers from 32-bit values
    - see also FromU64() for 64-bit unsigned integer values input

function TRawUTF8DynArrayFrom(const Values: array of RawUTF8): TRawUTF8DynArray;
    Quick helper to initialize a dynamic array of RawUTF8 from some constants
    - can be used e.g. as:
      MyArray := TRawUTF8DynArrayFrom(['a','b','c']);

function Trim(const S: RawUTF8): RawUTF8;
    Fast dedicated RawUTF8 version of Trim()
    - implemented using x86 asm, if possible
    - this Trim() is seldom used, but this RawUTF8 specific version is needed e.g. by Delphi 2009+, to
      avoid two unnecessary conversions into UnicodeString

function TrimControlChars(const text: RawUTF8; const controls: TSynAnsicharSet=[#0..' ']): RawUTF8;
    Returns the supplied text content, without any control char
    - a control char has an ASCII code #0 .. #32, i.e. text[]<=''
    - you can specify a custom char set to be excluded, if needed

procedure TrimCopy(const S: RawUTF8; start,count: PtrInt; out result: RawUTF8);
    Single-allocation (therefore faster) alternative to Trim(copy())

function TrimLeft(const S: RawUTF8): RawUTF8;
    Trims leading whitespace characters from the string by removing new line, space, and tab characters

function TrimLowerCase(V: RawUTF8): PUTF8Char;
    Trim first lowercase chars ('otDone' will return 'Done' e.g.)
    - return a PUTF8Char to avoid any memory allocation

function TrimLowerCaseShort(V: PShortString): RawUTF8;
    Trim first lowercase chars ('otDone' will return 'Done' e.g.)
    - return an RawUTF8 string: enumeration names are pure 7bit ANSI with Delphi 7 to 2007, and
      UTF-8 encoded with Delphi 2009+
procedure TrimLeftLowerCaseToShort(V: PShortString; out result: ShortString); overload;

Trim first lowercase chars ('otDone' will return 'Done' e.g.)
- return a shortstring: enumeration names are pure 7bit ANSI with Delphi 7 to 2007, and UTF-8 encoded with Delphi 2009+

function TrimLeftLowerCaseToShort(V: PShortString): ShortString; overload;

Trim first lowercase chars ('otDone' will return 'Done' e.g.)
- return a shortstring: enumeration names are pure 7bit ANSI with Delphi 7 to 2007, and UTF-8 encoded with Delphi 2009+

function TrimRight(const S: RawUTF8): RawUTF8;

Trims trailing whitespace characters from the string by removing trailing newline, space, and tab characters

function TruncTo2Digits(Value: Currency): Currency;

Truncate a Currency value to only 2 digits
- implementation will use fast Int64 math to avoid any precision loss due to temporary floating-point conversion

function TruncTo2Digits64(Value: Int64): Int64;

Truncate a Currency value, stored as Int64, to only 2 digits
- implementation will use fast Int64 math to avoid any precision loss due to temporary floating-point conversion

procedure TruncTo2DigitsCurr64(var Value: Int64);

Truncate a Currency value, stored as Int64, to only 2 digits
- implementation will use fast Int64 math to avoid any precision loss due to temporary floating-point conversion

function TryEncodeDate(Year, Month, Day: cardinal; out Date: TDateTime): Boolean;

Our own fast version of the corresponding low-level RTL function

function TypeInfoToHash(aTypeInfo: pointer): cardinal;

Compute a crc32c-based hash of the RTTI for a managed given type
- can be used to ensure that the RecordSave/TDynArray.SaveTo binary layout is compatible across executables, even between FPC and Delphi
- will ignore the type names, but will check the RTTI type kind and any nested fields (for records or arrays) - for a record/object type, will use TTextWriter.RegisterCustomJSONSerializerFromText definition, if available

function TypeInfoToName(aTypeInfo: pointer): RawUTF8; overload;

Retrieve the type name from its low-level RTTI

procedure TypeInfoToName(aTypeInfo: pointer; var result: RawUTF8; const default: RawUTF8=''); overload;

Retrieve the type name from its low-level RTTI

procedure TypeInfoToQualifiedName(aTypeInfo: pointer; var result: RawUTF8; const default: RawUTF8='');

Retrieve the unit name and type name from its low-level RTTI
function TypeInfoToRttiType(aTypeInfo: pointer): TJSONCustomParserRTTIType;
  // Recognize a simple type from a supplied type information
  // first try by name via TJSONCustomParserRTTI.TypeNameToSimpleRTTIType, then from RTTI via
  // TJSONCustomParserRTTI TypeInfoToSimpleRTTIType
  // will return ptCustom for any unknown type

function UCS4ToUTF8(ucs4: cardinal; Dest: PUTF8Char): integer;
  // UTF-8 encode one UCS4 character into Dest
  // return the number of bytes written into Dest (i.e. from 1 up to 6)
  // this method DOES handle UTF-16 surrogate pairs

function UInt2DigitsToShort(Value: byte): TShort4;
  // Creates a 2 digits short string from a 0..99 value
  // using TShort4 as returned string would avoid a string allocation on heap
  // could be used e.g. as parameter to FormatUTF8()

function UInt2DigitsToShortFast(Value: byte): TShort4;
  // Creates a 2 digits short string from a 0..99 value
  // won't test Value>99 as UInt2DigitsToShort()

function UInt3DigitsToUTF8(Value: Cardinal): RawUTF8;
  // Creates a 3 digits string from a 0..999 value
  // consider using UInt3DigitsToShort() to avoid temporary memory allocation, e.g. when used as
  // FormatUTF8() parameter

function UInt64ToUtf8(Value: QWord; var result: RawUTF8);
  // Fast RawUTF8 version of IntToStr(), with proper QWord conversion
function UnCamelCase(D, P: PUTF8Char): integer; overload;

*Convert a CamelCase string into a space separated one*
- 'OnLine' will return 'On line' e.g., and 'OnMyLINE' will return 'On my LINE'
- will handle capital words at the beginning, middle or end of the text, e.g. 'KLMFlightNumber' will return 'KLM flight number' and 'GoodBBCProgram' will return 'Good BBC program'
- will handle a number at the beginning, middle or end of the text, e.g. 'Email12' will return 'Email 12'
- return the char count written into D
- D^ and P^ are expected to be UTF-8 encoded: enumeration and property names are pure 7bit ANSI with Delphi 7 to 2007, and UTF-8 encoded with Delphi 2009+
- '_'.char is transformed into ' -'
- '__' chars are transformed into ': '

function UnCamelCase(const S: RawUTF8): RawUTF8; overload;

*Convert a CamelCase string into a space separated one*
- 'OnLine' will return 'On line' e.g., and 'OnMyLINE' will return 'On my LINE'
- will handle capital words at the beginning, middle or end of the text, e.g. 'KLMFlightNumber' will return 'KLM flight number' and 'GoodBBCProgram' will return 'Good BBC program'
- will handle a number at the beginning, middle or end of the text, e.g. 'Email12' will return 'Email 12'
- '_'.char is transformed into ' -'
- '__' chars are transformed into ': '
- return an RawUTF8 string: enumeration names are pure 7bit ANSI with Delphi 7 to 2007, and UTF-8 encoded with Delphi 2009+

function UnicodeBufferToString(source: PWideChar): string;

*Convert an Unicode buffer into a generic VCL string*

procedure UnicodeBufferToWinAnsi(source: PWideChar; out Dest: WinAnsiString);

*Convert an Unicode buffer into a WinAnsi (code page 1252) string*

function UniqueRawUTF8(var UTF8: RawUTF8): pointer;

*Equivalence to @UTF8[1] expression to ensure a RawUTF8 variable is unique*
- will ensure that the string recfcount is 1, and return a pointer to the text
- under FPC, @UTF8[1] does not call UniqueString() as it does with Delphi
- if UTF8 is a constant (recfcount=-1), will create a temporary copy in heap

procedure UniqueRawUTF8ZeroToTilde(var UTF8: RawUTF8; MaxSize: integer=maxInt);

*Will fast replace all #0 chars as ~*
- could be used after UniqueRawUTF8() on a in-placed modified JSON buffer, in which all values have been ended with #0
- you can optionally specify a maximum size, in bytes (this won't reallocate the string, but just add a #0 at some point in the UTF8 buffer)
- could allow logging of parsed input e.g. after an exception

function UnixMSTimePeriodToString(const UnixMSTime: TUnixMSTime; FirstTimeChar: AnsiChar='T'): RawUTF8;

*Delphi 2007 is buggy as hell convert some millisecond-based c-encoded time to the ISO 8601 text layout, as time or date elapsed period*
- this function won't add the Unix epoch 1/1/1970 offset to the timestamp
- returns 'Thh:mm:ss' or 'YYYY-MM-DD' format, depending on the supplied value
function UnixMSTimeToDateTime(const UnixMSTime: TUnixMSTime): TDateTime;
Convert a millisecond-based c-encoded time (from Unix epoch 1/1/1970) as TDateTime

function UnixMSTimeToFileShort(const UnixMSTime: TUnixMSTime): TShort16;
Convert some millisecond-based c-encoded time (from Unix epoch 1/1/1970) to a small text layout, trimming to the second resolution, perfect e.g. for naming a local file
- use 'YMMDDHHMMSS' format so year is truncated to last 2 digits, expecting a date > 1999 (a current date would be fine)

function UnixMSTimeToString(const UnixMSTime: TUnixMSTime; Expanded: boolean=true; FirstTimeChar: AnsiChar='T'; const TZD: RawUTF8=''): RawUTF8;
Convert some millisecond-based c-encoded time (from Unix epoch 1/1/1970) to the ISO 8601 text layout, including milliseconds
- i.e. 'YYYY-MM-DDThh:mm:ss.sssZ' or 'YYYYMMDDThhmmss.sssZ' format
- TZD is the ending time zone designator ('', 'Z' or '+hh:mm' or '-hh:mm')

function UnixMSTimeUTC: TUnixMSTime;
Returns the current UTC date/time as a millisecond-based c-encoded time
- i.e. current number of milliseconds elapsed since Unix epoch 1/1/1970
- faster and more accurate than NowUTC or GetTickCount64, on Windows or Unix
- will use e.g. fast clock_gettime(CLOCK_REALTIME_COARSE) under Linux, or GetSystemTimeAsFileTime/GetSystemTimePreciseAsFileTime under Windows - the later being more accurate, but slightly slower than the former, so you may consider using UnixMSTimeUTCFast on Windows if its 10-16ms accuracy is enough

function UnixMSTimeUTCFast: TUnixMSTime;
Returns the current UTC date/time as a millisecond-based c-encoded time under Linux/POSIX, is the very same than UnixMSTimeUTC under Windows 8+, will call GetSystemTimeAsFileTime instead of GetSystemTimePreciseAsFileTime, which has higher precision, but is slower
- prefer it under Windows, if a dozen of ms resolution is enough for your task

function UnixTimePeriodToString(const UnixTime: TUnixTime; FirstTimeChar: AnsiChar='T'): RawUTF8;
Delphi 2007 is buggy as hell convert some second-based c-encoded time to the ISO 8601 text layout, either as time or date elapsed period
- this function won't add the Unix epoch 1/1/1970 offset to the timestamp
- returns 'Thh:mm:ss' or 'YYYY-MM-DD' format, depending on the supplied value

function UnixTimeToDateTime(const UnixTime: TUnixTime): TDateTime;
Convert a second-based c-encoded time as TDateTime
- i.e. number of seconds elapsed since Unix epoch 1/1/1970 into TDateTime

procedure UnixTimeToFileShort(const UnixTime: TUnixTime; out result: TShort16); overload;
Convert some second-based c-encoded time (from Unix epoch 1/1/1970) to a small text layout, perfect e.g. for naming a local file
- use 'YMMDDHHMMSS' format so year is truncated to last 2 digits, expecting a date > 1999 (a current date would be fine)
function UnixTimeToFileShort(const UnixTime: TUnixTime): TShort16; overload;
Convert some second-based c-encoded time (from Unix epoch 1/1/1970) to a small text layout,
perfect e.g. for naming a local file.
- use 'YYMMDTHHHmss' format so year is truncated to last 2 digits, expecting a date > 1999 (a
current date would be fine)

function UnixTimeToString(const UnixTime: TUnixTime; Expanded: boolean=true;
FirstTimeChar: AnsiChar='T'): RawUTF8;
Convert some second-based c-encoded time (from Unix epoch 1/1/1970) to the ISO 8601 text
layout
- use 'YYYYMMDDThhmmss' format if not Expanded
- use 'YYYY-MM-DDThh:mm:ss' format if Expanded

function UnixTimeUTC: TUnixTime;
Returns the current UTC date/time as a second-based c-encoded time
- i.e. current number of seconds elapsed since Unix epoch 1/1/1970
- faster than NowUTC or GetTickCount64, on Windows or Unix platforms (will use e.g. fast
clock_gettime(CLOCK_REALTIME_COARSE) under Linux, or GetSystemTimeAsFileTime under
Windows)
- returns a 64-bit unsigned value, so is "Year2038bug" free

function UnQuotedSQLSymbolName(const ExternalDBSymbol: RawUTF8): RawUTF8;
Unquote a SQL-compatible symbol name
- e.g. '[symbol]' -> 'symbol' or '"symbol"' -> 'symbol'

function UnQuoteSQLString(const Value: RawUTF8): RawUTF8;
Unquote a SQL-compatible string

function UnQuoteSQLStringVar(P: PUTF8Char; out Value: RawUTF8): PUTF8Char;
Unquote a SQL-compatible string
- the first character in P^ must be either ' or " then internal double quotes are transformed into
single quotes
- 'text " end' -> text ' end
- "text " end" -> text " end
- returns nil if P doesn't contain a valid SQL string
- returns a pointer just after the quoted text otherwise

procedure UnSetBit(var Bits; aIndex: PtrInt);
Unset/clear a particular bit into a bit array
- this function can't be inlined, whereas UnSetBitPtr() function can

procedure UnSetBit64(var Bits: Int64; aIndex: PtrInt);
Unset/clear a particular bit into a 64-bit integer bits (max aIndex is 63)

procedure UnSetBitPtr(Bits: pointer; aIndex: PtrInt);
Unset/clear a particular bit into a bit array
- UnSetBit() can't be inlined, whereas this pointer-oriented function can

procedure UpdateIniEntry(var Content: RawUTF8; const Section,Name,Value: RawUTF8);
Update a Name= Value in a [Section] of a INI RawUTF8 Content
- this function scans and update the Content memory buffer, and is therefore very fast (no
temporary TMemIniFile is created)
- if Section equals ", update the Name= value before any [Section]
procedure UpdateIniEntryFile(const FileName: TFileName; const Section, Name, Value: RawUTF8);

  Update a Name= Value in a [Section] of a .INI file
  - if Section equals ", use the Name= value before any [Section]
  - use internally fast UpdateIniEntry() function above

function UpdateIniNameValue(var Content: RawUTF8; const Name, UpperName, NewValue: RawUTF8): boolean;

  Replace a value from a given set of name=value lines
  - expect UpperName as 'UPPERNAME=', otherwise returns false
  - if no UPPERNAME= entry was found, then Name+NewValue is added to Content
  - a typical use may be:
    UpdateIniNameValue(headers,HEADER_CONTENT_TYPE,HEADER_CONTENT_TYPE_UPPER,contenttype);

function UpperCase(const S: RawUTF8): RawUTF8;

  Fast conversion of the supplied text into uppercase
  - this will only convert 'a'..'z' into 'A'..'Z' (no NormToUpper use), and will therefore be correct with
    true UTF-8 content, but only for 7 bit

procedure UpperCaseCopy(const Source: RawUTF8; var Dest: RawUTF8); overload;

  Fast conversion of the supplied text into uppercase
  - this will only convert 'a'..'z' into 'A'..'Z' (no NormToUpper use), and will therefore be correct with
    true UTF-8 content, but only for 7 bit

procedure UpperCaseCopy(Text: PUTF8Char; Len: PtrInt; var result: RawUTF8); overload;

  Fast conversion of the supplied text into uppercase
  - this will only convert 'a'..'z' into 'A'..'Z' (no NormToUpper use), and will therefore be correct with
    true UTF-8 content, but only for 7 bit

procedure UpperCaseSelf(var S: RawUTF8);

  Fast in-place conversion of the supplied variable text into uppercase
  - this will only convert 'a'..'z' into 'A'..'Z' (no NormToUpper use), and will therefore be correct with
    true UTF-8 content, but only for 7 bit

function UpperCaseU(const S: RawUTF8): RawUTF8;

  Fast conversion of the supplied text into 8 bit uppercase
  - this will not only convert 'a'..'z' into 'A'..'Z', but also accentuated latin characters ('e' acute into 'E'
    e.g.), using NormToUpper[] array
  - it will therefore decode the supplied UTF-8 content to handle more than 7 bit of ascii characters
    (so this function is dedicated to WinAnsi code page 1252 characters set)

function UpperCaseUnicode(const S: RawUTF8): RawUTF8;

  Accurate conversion of the supplied UTF-8 content into the corresponding upper-case Unicode
  characters
  - this version will use the Operating System API, and will therefore be much slower than
    UpperCase/UpperCaseU versions, but will handle all kind of unicode characters

function UpperCopy(dest: PAnsiChar; const source: RawUTF8): PAnsiChar;

  Copy source into dest^ with 7 bits upper case conversion
  - returns final dest pointer
  - will copy up to the source buffer end: so Dest^ should be big enough - which will the case e.g. if
    Dest := pointer(source)
function UpperCopy255(dest: PAnsiChar; const source: RawUTF8): PAnsiChar; overload;
    Copy source into a 256 chars dest^ buffer with 7 bits upper case conversion
    - used internally for short keys match or case-insensitive hash
    - returns final dest pointer
    - will copy up to 255 AnsiChar (expect the dest buffer to be defined e.g. as array[byte] of AnsiChar on the caller stack)

function UpperCopy255BufPas(dest: PAnsiChar; source: PUTF8Char; sourceLen: PtrInt): PAnsiChar;
    Copy source^ into a 256 chars dest^ buffer with 7 bits upper case conversion
    - used internally for short keys match or case-insensitive hash
    - this version is written in optimized pascal
    - you should not have to call this function, but rely on UpperCopy255Buf()
    - returns final dest pointer
    - will copy up to 255 AnsiChar (expect the dest buffer to be defined e.g. as array[byte] of AnsiChar on the caller stack)

function UpperCopy255BufSSE42(dest: PAnsiChar; source: PUTF8Char; sourceLen: PtrInt): PAnsiChar;
    SSE 4.2 version of UpperCopy255Buf()
    - copy source^ into a 256 chars dest^ buffer with 7 bits upper case conversion
    - please note that this optimized version may read up to 15 bytes beyond the string; this is rarely a problem but it may generate protection violations, which could trigger fatal SIGABRT or SIGSEGV on Posix system
    - could be used instead of UpperCopy255Buf() when you are confident about your dest/source input buffers, checking if cfSSE42 in CpuFeatures

function UpperCopy255W(dest: PAnsiChar; source: PWideChar; L: integer): PAnsiChar;
    Copy WideChar source into dest^ with upper case conversion
    - used internally for short keys match or case-insensitive hash
    - returns final dest pointer
    - will copy up to 255 AnsiChar (expect the dest buffer to be array[byte] of AnsiChar)

function UpperCopy255W(dest: PAnsiChar; const source: SynUnicode): PAnsiChar;
    Copy WideChar source into dest^ with upper case conversion
    - used internally for short keys match or case-insensitive hash
    - returns final dest pointer
    - will copy up to 255 AnsiChar (expect the dest buffer to be array[byte] of AnsiChar)

function UpperCopyShort(dest: PAnsiChar; const source: shortstring): PAnsiChar;
    Copy source into dest^ with 7 bits upper case conversion
    - returns final dest pointer
    - this special version expect source to be a shortstring

function UpperCopyWin255(dest: PWinAnsiChar; const source: RawUTF8): PWinAnsiChar;
    Copy source into dest^ with WinAnsi 8 bits upper case conversion
    - used internally for short keys match or case-insensitive hash
    - returns final dest pointer
    - will copy up to 255 AnsiChar (expect the dest buffer to be array[byte] of AnsiChar)
function UrlDecode(U: PUTF8Char): RawUTF8; overload;
Decode a string compatible with URI encoding into its original value

function UrlDecode(const s: RawUTF8; i: PtrInt=1; len: PtrInt=-1): RawUTF8; overload;
- you can specify the decoding range (as in copy(s,i,len) function)

function UrlDecodeCardinal(U: PUTF8Char; const Upper: RawUTF8; var Value: Cardinal; Next: PPUTF8Char=nil): boolean;
Decode a specified parameter compatible with URI encoding into its original cardinal numerical value
- UrlDecodeCardinal('offset=20&where=LastName%3D%27M%C3%B4net%27','OFFSET=',O,@Next)
  will return Next^='where=...' and O=20
  - if Upper is not found, Value is not modified, and result is FALSE
  - if Upper is found, Value is modified with the supplied content, and result is TRUE

function UrlDecodeDouble(U: PUTF8Char; const Upper: RawUTF8; var Value: double; Next: PPUTF8Char=nil): boolean;
Decode a specified parameter compatible with URI encoding into its original floating-point value
- UrlDecodeDouble('price=20.45&where=LastName%3D%27M%C3%B4net%27','PRICE=',P,@Next)
  will return Next^='where=...' and P=20.45
  - if Upper is not found, Value is not modified, and result is FALSE
  - if Upper is found, Value is modified with the supplied content, and result is TRUE

function UrlDecodeExtended(U: PUTF8Char; const Upper: RawUTF8; var Value: TSynExtended; Next: PPUTF8Char=nil): boolean;
Decode a specified parameter compatible with URI encoding into its original floating-point value
- UrlDecodeExtended('price=20.45&where=LastName%3D%27M%C3%B4net%27','PRICE=',P,@Next)
  will return Next^='where=...' and P=20.45
  - if Upper is not found, Value is not modified, and result is FALSE
  - if Upper is found, Value is modified with the supplied content, and result is TRUE

function UrlDecodeInt64(U: PUTF8Char; const Upper: RawUTF8; var Value: Int64; Next: PPUTF8Char=nil): boolean;
Decode a specified parameter compatible with URI encoding into its original Int64 numerical value
- UrlDecodeInt64('offset=20&where=LastName%3D%27M%C3%B4net%27','OFFSET=',O,@Next)
  will return Next^='where=...' and O=20
  - if Upper is not found, Value is not modified, and result is FALSE
  - if Upper is found, Value is modified with the supplied content, and result is TRUE

function UrlDecodeInteger(U: PUTF8Char; const Upper: RawUTF8; var Value: integer; Next: PPUTF8Char=nil): boolean;
Decode a specified parameter compatible with URI encoding into its original integer numerical value
- UrlDecodeInteger('offset=20&where=LastName%3D%27M%C3%B4net%27','OFFSET=',O,@Next)
  will return Next^='where=...' and O=20
  - if Upper is not found, Value is not modified, and result is FALSE
  - if Upper is found, Value is modified with the supplied content, and result is TRUE
functiongetUrlDecodeNeedParameters(U, CSVNames: PUTF8Char): boolean;

Returns TRUE if all supplied parameters do exist in the URI encoded text
- CSVNames parameter shall provide as a CSV list of names
- e.g. getUrlDecodeNeedParameters('price=20.45&where=LastName%3D','price,where') will return
  TRUE

function getUrlDecodeNextName(U: PUTF8Char; out Name: RawUTF8): PUTF8Char;

Decode a URI-encoded Name from an input buffer
- decoded value is set in Name out variable
- returns a pointer just after the decoded name, after the '='
- returns nil if there was no name=... pattern in U

function getUrlDecodeNextNameValue(U: PUTF8Char; var Name, Value: RawUTF8): PUTF8Char;

Decode the next Name=Value&... pair from input URI
- Name is returned directly (should be plain ASCII 7 bit text)
- Value is returned after URI decoding (from %.. patterns)
- if a pair is decoded, return a PUTF8Char pointer to the next pair in the input buffer, or points to
  #0 if all content has been processed
- if a pair is not decoded, return nil

function getUrlDecodeNextValue(U: PUTF8Char; out Value: RawUTF8): PUTF8Char;

Decode a URI-encoded Value from an input buffer
- decoded value is set in Value out variable
- returns a pointer just after the decoded value (may points e.g. to
  \0 or '\&') - it is up to the caller to continue the process or not

function getUrlDecodeValue(U: PUTF8Char; const Upper: RawUTF8; var Value: RawUTF8; Next: PUTF8Char=nil): boolean;

Decode a specified parameter compatible with URI encoding into its original textual value
- getUrlDecodeValue('select=%2A&where=LastName%3D%27M%C3%B4net%27','SELECT=', V, @Next)
  will return Next^='where=...' and V='*'
- if Upper is not found, Value is not modified, and result is FALSE
- if Upper is found, Value is modified with the supplied content, and result is TRUE

function getUrlEncode(const svar: RawUTF8): RawUTF8; overload;

Encode a string to be compatible with URI encoding

function getUrlEncode(const NameValuePairs: array of const): RawUTF8; overload;

Encode supplied parameters to be compatible with URI encoding
- parameters must be supplied two by two, as Name,Value pairs, e.g.
  url := getUrlEncode(['select','*','where','ID=12','offset',23,'object',aObject]);
- parameters names should be plain ASCII-7 RFC compatible identifiers (0..9a..Z_.~), otherwise
  their values are skipped
- parameters values can be either textual, integer or extended, or any TObject
- TObject serialization into UTF-8 will be processed by the ObjectToJSON() function

function getUrlEncode(Text: PUTF8Char): RawUTF8; overload;

Encode a string to be compatible with URI encoding
function UrlEncodeJsonObject(const URIName, ParametersJSON: RawUTF8; const PropNamesToIgnore: array of RawUTF8; IncludeQueryDelimiter: Boolean=true): RawUTF8; overload;

Encode a JSON object UTF-8 buffer into URI parameters
- you can specify property names to ignore during the object decoding
- you can omit the leading query delimiter ('?') by setting IncludeQueryDelimiter=false
- overloaded function which will make a copy of the input JSON before parsing

function UrlEncodeJsonObject(const URIName: RawUTF8; ParametersJSON: PUTF8Char; const PropNamesToIgnore: array of RawUTF8; IncludeQueryDelimiter: Boolean=true): RawUTF8; overload;

Encode a JSON object UTF-8 buffer into URI parameters
- you can specify property names to ignore during the object decoding
- you can omit the leading query delimiter ('?') by setting IncludeQueryDelimiter=false
- warning: the ParametersJSON input buffer will be modified in-place

function UTF16CharToUtf8(Dest: PUTF8Char; var Source: PWord): integer;

UTF-8 encode one UTF-16 encoded UCS4 character into Dest
- return the number of bytes written into Dest (i.e. from 1 up to 6)
- Source will contain the next UTF-16 character
- this method DOES handle UTF-16 surrogate pairs

function Utf8DecodeToRawUnicode(P: PUTF8Char; L: integer): RawUnicode; overload;

Convert a UTF-8 encoded buffer into a RawUnicode string
- if L is 0, L is computed from zero terminated P buffer
- RawUnicode is ended by a WideChar(#0)
- faster than System.Utf8Decode() which uses slow widestrings

function Utf8DecodeToRawUnicode(const S: RawUTF8): RawUnicode; overload;

Convert a UTF-8 string into a RawUnicode string

function Utf8DecodeToRawUnicodeUI(const S: RawUTF8; DestLen: PInteger=nil): RawUnicode; overload;

Convert a UTF-8 string into a RawUnicode string
- this version doesn't resize the length of the result RawUnicode and is therefore useful before a Win32 Unicode API call (with nCount=-1)
- if DestLen is not nil, the resulting length (in bytes) will be stored within

function Utf8DecodeToString(P: PUTF8Char; L: integer; var result: string); overload;

Convert any UTF-8 encoded buffer into a generic VCL Text
- it's preferred to use TLanguageFile.UTF8ToString() in mORMoti18n, which will handle full i18n of your application
- it will work as is with Delphi 2009+ (direct unicode conversion)
- under older version of Delphi (no unicode), it will use the current RTL codepage, as with WideString conversion (but without slow WideString usage)

procedure UTF8DecodeToString(P: PUTF8Char; L: integer; var result: string); overload;

Convert any UTF-8 encoded buffer into a generic VCL Text
function Utf8FirstLineToUnicodeLength(source: PUTF8Char): PtrInt;
    Calculate the UTF-16 Unicode characters count of the UTF-8 encoded first line
    - count may not match the UCS4 glyphs number, in case of UTF-16 surrogates
    - end the parsing at first #13 or #10 character

function UTF8IComp(u1, u2: PUTF8Char): PtrInt;
    Fast UTF-8 comparison using the NormToUpper[] array for all 8 bits values
    - this version expects u1 and u2 to be zero-terminated
    - this version will decode each UTF-8 glyph before using NormToUpper[]
    - current implementation handles UTF-16 surrogates

function UTF8ILComp(u1, u2: PUTF8Char; L1,L2: cardinal): PtrInt;
    Fast UTF-8 comparison using the NormToUpper[] array for all 8 bits values
    - this version expects u1 and u2 not to be necessary zero-terminated, but uses L1 and L2 as length
    for u1 and u2 respectively
    - use this function for SQLite3 collation (TSQLCollateFunc)
    - this version will decode the UTF-8 content before using NormToUpper[]
    - current implementation handles UTF-16 surrogates

function UTF8ToInt64(const text: RawUTF8; const default: Int64=0): Int64;
    Get the signed 64-bit integer value stored in a RawUTF8 string
    - returns the default value if the supplied text was not successfully converted into an Int64

function UTF8ToInteger(const value: RawUTF8; Default: PtrInt=0): PtrInt; overload;
    Get the signed 32-bit integer value stored in a RawUTF8 string
    - we use the PtrInt result type, even if expected to be 32-bit, to use native CPU register size (don't want any 32-bit overflow here)

function UTF8ToInteger(const value: RawUTF8; Min,max: PtrInt; Default: PtrInt=0): PtrInt; overload;
    Get and check range of a signed 32-bit integer stored in a RawUTF8 string
    - we use the PtrInt result type, even if expected to be 32-bit, to use native CPU register size (don't want any 32-bit overflow here)

procedure Utf8ToRawUTF8(P: PUTF8Char; var result: RawUTF8);
    Direct conversion of a UTF-8 encoded zero terminated buffer into a RawUTF8 String

procedure UTF8ToShortString(var dest: shortstring; source: PUTF8Char);
    Direct conversion of a UTF-8 encoded buffer into a WinAnsi shortstring buffer

function UTF8ToString(const Text: RawUTF8): string;
    Convert any UTF-8 encoded String into a generic VCL Text
    - it's preferred to use TLanguageFile.UTF8ToString() in mORMoti18n, which will handle full i18n of your application
    - it will work as is with Delphi 2009+ (direct unicode conversion)
    - under older version of Delphi (no unicode), it will use the current RTL codepage, as with WideString conversion (but without slow WideString usage)

procedure UTF8ToSynUnicode(Text: PUTF8Char; Len: PtrInt; var result: SynUnicode);
    Convert any UTF-8 encoded buffer into a generic SynUnicode Text
procedure UTF8ToSynUnicode(const Text: RawUTF8; var result: SynUnicode); overload;
Convert any UTF-8 encoded String into a generic SynUnicode Text

function UTF8ToSynUnicode(const Text: RawUTF8): SynUnicode; overload;
Convert any UTF-8 encoded String into a generic SynUnicode Text

function Utf8ToUnicodeLength(source: PUTF8Char): PtrUInt;
Calculate the UTF-16 Unicode characters count, UTF-8 encoded in source^
- count may not match the UCS4 glyphs number, in case of UTF-16 surrogates
- faster than System.UTF8ToUnicode with dest=nil

function UTF8ToWideChar(dest: PWideChar; source: PUTF8Char; sourceBytes: PtrInt=0; NoTrailingZero: boolean=false): PtrInt; overload;
Convert an UTF-8 encoded text into a WideChar (UTF-16) buffer
- faster than System.UTF8ToUnicode
- sourceBytes can be 0, therefore length is computed from zero terminated source
- enough place must be available in dest buffer (guess is sourceBytes*3+2)
- a WideChar(#0) is added at the end (if something is written) unless NoTrailingZero is TRUE
- returns the BYTE count written in dest, excluding the ending WideChar(#0)

function UTF8ToWideChar(dest: PWideChar; source: PUTF8Char; MaxDestChars, sourceBytes: PtrInt; NoTrailingZero: boolean=false): PtrInt; overload;
Convert an UTF-8 encoded text into a WideChar (UTF-16) buffer
- faster than System.UTF8ToUnicode
- this overloaded function expect a MaxDestChars parameter
- sourceBytes can not be 0 for this function
- enough place must be available in dest buffer (guess is sourceBytes*3+2)
- a WideChar(#0) is added at the end (if something is written) unless NoTrailingZero is TRUE
- returns the BYTE COUNT (not WideChar count) written in dest, excluding the ending WideChar(#0)

procedure UTF8ToWideString(Text: PUTF8Char; Len: PtrInt; var result: WideString); overload;
Convert any UTF-8 encoded String into a generic WideString Text

function UTF8ToWideString(const Text: RawUTF8): WideString; overload;
Convert any UTF-8 encoded String into a generic WideString Text

procedure UTF8ToWideString(const Text: RawUTF8; var result: WideString); overload;
Convert any UTF-8 encoded String into a generic WideString Text

function Utf8ToWinAnsi(const S: RawUTF8): WinAnsiString; overload;
Direct conversion of a UTF-8 encoded string into a WinAnsi String

function Utf8ToWinAnsi(P: PUTF8Char): WinAnsiString; overload;
Direct conversion of a UTF-8 encoded zero terminated buffer into a WinAnsi String

function UTF8ToWinPChar(dest: PAnsiChar; source: PUTF8Char; count: integer): integer;
Direct conversion of a UTF-8 encoded buffer into a WinAnsi PAnsiChar buffer
function Utf8TruncatedLength(text: PAnsiChar; textlen,maxBytes: PtrUInt): PtrInt;

Compute the truncated length of the supplied UTF-8 value if it exceeds the specified bytes count
- this function will ensure that the returned content will contain only valid UTF-8 sequence, i.e.
  will trim the whole trailing UTF-8 sequence
- returns maxUTF8 if text was not truncated, or the number of fitting bytes

function Utf8TruncatedLength(const text: RawUTF8; maxBytes: PtrUInt): PtrInt;

Compute the truncated length of the supplied UTF-8 value if it exceeds the specified bytes count
- this function will ensure that the returned content will contain only valid UTF-8 sequence, i.e.
  will trim the whole trailing UTF-8 sequence
- returns maxUTF8 if text was not truncated, or the number of fitting bytes

function Utf8TruncateToLength(var text: RawUTF8; maxBytes: PtrUInt): boolean;

Will truncate the supplied UTF-8 value if its length exceeds the specified bytes count
- this function will ensure that the returned content will contain only valid UTF-8 sequence, i.e.
  will trim the whole trailing UTF-8 sequence
- returns FALSE if text was not truncated, TRUE otherwise

function Utf8TruncateToUnicodeLength(var text: RawUTF8; maxUtf16: integer): boolean;

Will truncate the supplied UTF-8 value if its length exceeds the specified UTF-16 Unicode characters count
- count may not match the UCS4 glyphs number, in case of UTF-16 surrogates
- returns FALSE if text was not truncated, TRUE otherwise

function UTF8UpperCopy(Dest, Source: PUTF8Char; SourceChars: Cardinal): PUTF8Char;

Copy WideChar source into dest^ with upper case conversion, using the NormToUpper[] array for
all 8 bits values, encoding the result as UTF-8
- returns final dest pointer
- current implementation handles UTF-16 surrogates

function UTF8UpperCopy255(dest: PAnsiChar; const source: RawUTF8): PUTF8Char;

Copy WideChar source into dest^ with upper case conversion, using the NormToUpper[] array for
all 8 bits values, encoding the result as UTF-8
- returns final dest pointer
- will copy up to 255 AnsiChar (expect the dest buffer to be array[byte] of AnsiChar), with UTF-8
  encoding

function ValuesToVariantDynArray(const items: array of const): TVariantDynArray;

Convert an open array list into a dynamic array of variants
- will use a TDocVariantData temporary storage

function VarDataIsEmptyOrNull(VarData: pointer): Boolean;

Same as VarIsEmpty(PVariant(V)^) or VarIsEmpty(PVariant(V)^), but faster
- we also discovered some issues with FPC's Variants unit, so this function may be used even in
  end-user cross-compiler code

function VariantCompare(const V1,V2: variant): PtrInt;

TVariantCompare-compatible case-sensitive comparison function
- just a wrapper around SortDynArrayVariantComp(caseInsensitive=false)
**function** VariantCompareI(const V1,V2: variant): PTrInt;

TVariantCompare-compatible case-insensitive comparison function
- just a wrapper around SortDynArrayVariantComp(caseInsensitive=true)

**procedure** VariantDynArrayClear(var Value: TVariantDynArray);

Faster alternative to Finalize(aVariantDynArray)
- this function will take account and optimize the release of a dynamic array of custom variant types values
- for instance, an array of TDocVariant will be optimized for speed

**function** VariantDynArrayToJSON(const V: TVariantDynArray): RawUTF8;

Convert a dynamic array of variants into its JSON serialization
- will use a TDocVariantData temporary storage

**function** VariantEquals(const V: Variant; const Str: RawUTF8; CaseSensitive: boolean=true): boolean; overload;

Fast comparison of a Variant and UTF-8 encoded String (or number)
- slightly faster than plain V=Str, which computes a temporary variant
- here Str="" equals unassigned, null or false
- if CaseSensitive is false, will use IdemPropNameU() for comparison

**function** VariantHash(const value: variant; CaseInsensitive: boolean; Hasher: THasher=nil): cardinal;

Crc32c-based hash of a variant value
- complex string types will make up to 255 uppercase characters conversion if CaseInsensitive is true
- you can specify your own hashing function if crc32c is not what you expect

**function** VariantHexDisplayToBin(const Hex: variant; Bin: PByte; BinBytes: integer): boolean;

Fast conversion from hexa chars, supplied as a variant string, into a binary buffer

**function** VariantLoad(const Bin: RawByteString; CustomVariantOptions: PDocVariantOptions): variant; overload;

Retrieve a variant value from our optimized binary serialization format
- follow the data layout as used by RecordLoad() or VariantSave() function
- return varEmpty if the Source buffer is incorrect
- just a wrapper around VariantLoad()
- how custom type variants are created can be defined via CustomVariantOptions

**function** VariantLoad(var Value: variant; Source: PAnsiChar; CustomVariantOptions: PDocVariantOptions; SourceMax: PAnsiChar=nil): PAnsiChar; overload;

Retrieve a variant value from our optimized binary serialization format
- follow the data layout as used by RecordLoad() or VariantSave() function
- return nil if the Source buffer is incorrect
- in case of success, return the memory buffer pointer just after the read content
- how custom type variants are created can be defined via CustomVariantOptions
function VariantLoadJSON(const JSON: RawUTF8; TryCustomVariants: PDocVariantOptions=nil; AllowDouble: boolean=false): variant; overload;

Retrieve a variant value from a JSON number or string
- follows TTextWriter.AddVariant() format (calls GetVariantFromJSON)
- will instantiate either an Integer, Int64, currency, double or string value (as RawUTF8), guessing the best numeric type according to the textual content, and string in all other cases, except TryCustomVariants points to some options (e.g. @JSON_OPTIONS[true] for fast instance) and input is a known object or array, either encoded as strict-JSON (i.e. {...} or [...]), or with some extended (e.g. BSON) syntax
- this overloaded procedure will make a temporary copy before JSON parsing and return the variant as result

function VariantLoadJSON(var Value: variant; JSON: PUTF8Char; EndOfObject: PUTF8Char=nil; TryCustomVariants: PDocVariantOptions=nil; AllowDouble: boolean=false): PUTF8Char; overload;

Retrieve a variant value from a JSON number or string
- follows TTextWriter.AddVariant() format (calls GetVariantFromJSON)
- will instantiate either an Integer, Int64, currency, double or string value (as RawUTF8), guessing the best numeric type according to the textual content, and string in all other cases, except TryCustomVariants points to some options (e.g. @JSON_OPTIONS[true] for fast instance) and input is a known object or array, either encoded as strict-JSON (i.e. {...} or [...]), or with some extended (e.g. BSON) syntax
- warning: the JSON buffer will be modified in-place during process - use a temporary copy or the overloaded functions with RawUTF8 parameter if you need to access it later

procedure VariantLoadJSON(var Value: Variant; const JSON: RawUTF8; TryCustomVariants: PDocVariantOptions=nil; AllowDouble: boolean=false); overload;

Retrieve a variant value from a JSON number or string
- follows TTextWriter.AddVariant() format (calls GetVariantFromJSON)
- will instantiate either an Integer, Int64, currency, double or string value (as RawUTF8), guessing the best numeric type according to the textual content, and string in all other cases, except TryCustomVariants points to some options (e.g. @JSON_OPTIONS[true] for fast instance) and input is a known object or array, either encoded as strict-JSON (i.e. {...} or [...]), or with some extended (e.g. BSON) syntax
- this overloaded procedure will make a temporary copy before JSON parsing and return the variant as result

function VariantSave(const Value: variant): RawByteString; overload;

Save a Variant content into a binary buffer
- will handle standard Variant types and custom types (serialized as JSON)
- will return "" in case of an invalid (not handled) Variant type
- just a wrapper around VariantSaveLength()+VariantSave()
- warning: will encode generic string fields as within the variant type itself: using this function between UNICODE and NOT UNICODE versions of Delphi, will probably fail - you have been warned!
**function** VariantSave(const Value: variant; Dest: PAnsiChar): PAnsiChar; overload;

*Save a Variant content into a destination memory buffer*
- Dest must be at least VariantSaveLength() bytes long
- will handle standard Variant types and custom types (serialized as JSON)
- will return nil in case of an invalid (not handled) Variant type
- will use a proprietary binary format, with some variable-length encoding of the string length
- warning: will encode generic string fields as within the variant type itself: using this function between UNICODE and NOT UNICODE versions of Delphi, will probably fail - you have been warned!

**procedure** VariantSaveJSON(const Value: variant; Escape: TTextWriterKind; var result: RawUTF8); overload;

*Save a variant value into a JSON content*
- follows the TTextWriter.AddVariant() and VariantLoadJSON() format
- is able to handle simple and custom variant types, for instance:
  ```pascal
  VariantSaveJSON(1.5)='1.5'
  VariantSaveJSON('test')='"test"'
  o := _Json(('[BSON: ["test", 5.05, 1986]]');
  VariantSaveJSON(o)=('[BSON: ["test", 5.05, 1986]]');
  VariantSaveJSON('"name":"John","doc":{"one":1,"two":
  _Arr(["one",2])}'])
  VariantSaveJSON(o)=('[name":"John","doc":{"one":1,"two":["one",2]}]')
  ``
- note that before Delphi 2009, any varString value is expected to be a RawUTF8 instance - which does make sense in the mORMot area

**function** VariantSaveJSONLength(const Value: variant; Escape: TTextWriterKind=twJSONEscape): integer;

*Compute the number of chars needed to save a variant value into a JSON content*
- follows the TTextWriter.AddVariant() and VariantLoadJSON() format
- this will be much faster than length(VariantSaveJSON()) for huge content
- note that before Delphi 2009, any varString value is expected to be a RawUTF8 instance - which does make sense in the mORMot area

**function** VariantToBoolean(const V: Variant; var Value: Boolean): boolean;

*Convert any numerical Variant into a boolean value*
- text content will return true after case-insensitive 'true' comparison
function VariantToCurrency(const V: Variant; var Value: currency): boolean;

Convert any numerical Variant into a fixed decimals floating point value

function VariantToDateTime(const V: Variant; var Value: TDateTime): boolean;

Convert any date/time Variant into a TDateTime value
- would handle varDate kind of variant, or use a string conversion and ISO-8601 parsing if possible

function VariantToDouble(const V: Variant; var Value: double): boolean;

Convert any numerical Variant into a floating point value

function VariantToDoubleDef(const V: Variant; const default: double=0): double;

Convert any numerical Variant into a floating point value

procedure VariantToInlineValue(const V: Variant; var result: RawUTF8);

Convert any Variant into a value encoded as with :(...:) inlined parameters in FormatUTF8(Format,Args,Params)

function VariantToInt64(const V: Variant; var Value: Int64): boolean;

Convert any numerical Variant into a 64-bit integer
- it will expect true numerical Variant and won't convert any string nor floating-pointer Variant, which will return FALSE and won't change the Value variable content

function VariantToInt64Def(const V: Variant; DefaultValue: Int64): Int64;

Convert any numerical Variant into a 64-bit integer
- it will expect true numerical Variant and won't convert any string nor floating-pointer Variant, which will return the supplied DefaultValue

function VariantToInteger(const V: Variant; var Value: integer): boolean;

Convert any numerical Variant into a 32-bit integer
- it will expect true numerical Variant and won't convert any string nor floating-pointer Variant, which will return FALSE and won't change the Value variable content

function VariantToIntegerDef(const V: Variant; DefaultValue: integer): integer; overload;

Convert any numerical Variant into an integer
- it will expect true numerical Variant and won't convert any string nor floating-pointer Variant, which will return the supplied DefaultValue

procedure VariantToRawByteString(const Value: variant; var Dest: RawByteString);

Convert back a RawByteString from a variant
- the supplied variant should have been created via a RawByteStringToVariant() function call

function VariantToString(const V: Variant): string;

Convert any Variant into a VCL string type
- expects any varString value to be stored as a RawUTF8
- prior to Delphi 2009, use VariantToString(aVariant) instead of string(aVariant) to safely retrieve a string=AnsiString value from a variant generated by our framework units - otherwise, you may loose encoded characters
- for Unicode versions of Delphi, there won't be any potential data loss, but this version may be slightly faster than a string(aVariant)
**procedure** VariantToUTF8(const V: Variant; var result: RawUTF8; var wasString: boolean); overload;

*Convert any Variant into UTF-8 encoded String*
- use VariantSaveJSON() instead if you need a conversion to JSON with custom parameters
- wasString is set if the V value was a text
- empty and null variants will be stored as 'null' text - as expected by JSON
- custom variant types (e.g. TDocVariant) will be stored as JSON

**function** VariantToUTF8(const V: Variant): RawUTF8; overload;

*Convert any Variant into UTF-8 encoded String*
- use VariantSaveJSON() instead if you need a conversion to JSON with custom parameters
- note: null will be returned as 'null'

**function** VariantToUTF8(const V: Variant; var Text: RawUTF8): boolean; overload;

*Convert any Variant into UTF-8 encoded String*
- use VariantSaveJSON() instead if you need a conversion to JSON with custom parameters
- returns TRUE if the V value was a text, FALSE if was not (e.g. a number)
- empty and null variants will be stored as 'null' text - as expected by JSON
- custom variant types (e.g. TDocVariant) will be stored as JSON

**function** VariantToVariantUTF8(const V: Variant): Variant;

*Convert any Variant into another Variant storing an RawUTF8 of the value*
- e.g. VariantToVariantUTF8('toto')='toto' and VariantToVariantUTF8(12)='12'

**procedure** VariantToVarRec(const V: variant; var result: TVarRec);

*Convert a variant to an open array (const Args: array of const) argument*
- will always map to a vtVariant kind of argument

**function** VarIs(const V: Variant; const VTypes: TVarDataTypes): Boolean;

*Allow to check for a specific set of TVarData.VType*

**function** VarIsEmptyOrNull(const V: Variant): Boolean;

*Same as VarIsEmpty(V) or VarIsEmpty(V), but faster*
- we also discovered some issues with FPC's Variants unit, so this function may be used even in end-user cross-compiler code

**function** VarIsVoid(const V: Variant): boolean;

*Fastcheck if a variant hold a value*
- varEmpty, varNull or a " string would be considered as void
- varBoolean=false or varDate=0 would be considered as void
- a TDocVariantData with Count=0 would be considered as void
- any other value (e.g. integer) would be considered as not void

**function** VarRecAsChar(const V: TVarRec): integer;

*Get an open array (const Args: array of const) character argument*
- only handle varChar and varWideChar kind of arguments
function VarRecToDouble(const V: TVarRec; out value: double): boolean;

Convert an open array (const Args: array of const) argument to a floating point value
- returns TRUE and set Value if the supplied argument is a number (e.g. vtInteger, vtInt64, vtCurrency or vtExtended)
- returns FALSE if the argument is not a number
- note that, due to a Delphi compiler limitation, cardinal values should be type-casted to Int64()
  (otherwise the integer mapped value will be converted)

procedure VarRecToInlineValue(const V: TVarRec; var result: RawUTF8);

Convert an open array (const Args: array of const) argument to a value encoded as with :(...):
  inlined parameters in FormatUTF8(Format,Args,Params)
- note that, due to a Delphi compiler limitation, cardinal values should be type-casted to Int64()
  (otherwise the integer mapped value will be converted)
- any supplied TObject instance will be written as their class name

function VarRecToInt64(const V: TVarRec; out value: Int64): boolean;

Convert an open array (const Args: array of const) argument to an Int64
- returns TRUE and set Value if the supplied argument is a vtInteger, vtInt64 or vtBoolean
- returns FALSE if the argument is not an integer
- note that, due to a Delphi compiler limitation, cardinal values should be type-casted to Int64()
  (otherwise the integer mapped value will be converted)

function VarRecToTempUTF8(const V: TVarRec; var Res: TTempUTF8): integer;

Convert an open array (const Args: array of const) argument to an UTF-8 encoded text, using a
  specified temporary buffer
- this function would allocate a RawUTF8 in TempRawUTF8 only if needed, but use the supplied
  Res.Temp[] buffer for numbers to text conversion - caller should ensure to make
  RawUTF8(TempRawUTF8) := " on the entry
- it would return the number of UTF-8 bytes, i.e. Res.Len
- note that, due to a Delphi compiler limitation, cardinal values should be type-casted to Int64()
  (otherwise the integer mapped value will be converted)
- any supplied TObject instance will be written as their class name

procedure VarRecToUTF8(const V: TVarRec; var result: RawUTF8; wasString: PBoolean=nil);

Convert an open array (const Args: array of const) argument to an UTF-8 encoded text
- note that, due to a Delphi compiler limitation, cardinal values should be type-casted to Int64()
  (otherwise the integer mapped value will be converted)
- any supplied TObject instance will be written as their class name

function VarRecToUTF8IsString(const V: TVarRec; var value: RawUTF8): boolean;

Convert an open array (const Args: array of const) argument to an UTF-8 encoded text, returning
  FALSE if the argument was not a string value

procedure VarRecToVariant(const V: TVarRec; var result: variant); overload;

Convert an open array (const Args: array of const) argument to a variant
- note that, due to a Delphi compiler limitation, cardinal values should be type-casted to Int64()
  (otherwise the integer mapped value will be converted)
function VarRecToVariant(const V: TVarRec): variant; overload;
    Convert an open array (const Args: array of const) argument to a variant
    - note that, due to a Delphi compiler limitation, cardinal values should be type-casted to Int64()
    (otherwise the integer mapped value will be converted)

function VarStringOrNull(const v: RawUTF8): variant;
    Returns a supplied string as variant, or null if v is void ("")

function WideCharToUtf8(Dest: PUTF8Char; aWideChar: PtrUInt): integer;
    UTF-8 encode one UTF-16 character into Dest
    - return the number of bytes written into Dest (i.e. 1,2 or 3)
    - this method does NOT handle UTF-16 surrogate pairs

function WideCharToWinAnsi(wc: cardinal): integer;
    Conversion of a wide char into a WinAnsi (CodePage 1252) char index
    - return -1 for an unknown WideChar in code page 1252

function WideCharToWinAnsiChar(wc: cardinal): AnsiChar;
    Conversion of a wide char into a WinAnsi (CodePage 1252) char
    - return '?' for an unknown WideChar in code page 1252

function WideStringToUTF8(const aText: WideString): RawUTF8;
    Convert a WideString into a UTF-8 string

function WideStringToWinAnsi(const Wide: WideString): WinAnsiString;
    Convert a WideString into a WinAnsi (code page 1252) string

function WinAnsiBufferToUtf8(Dest: PUTF8Char; Source: PAnsiChar; SourceChars: Cardinal): PUTF8Char;
    Direct conversion of a WinAnsi PAnsiChar buffer into a UTF-8 encoded buffer
    - Dest^ buffer must be reserved with at least SourceChars*3
    - call internally WinAnsiConvert fast conversion class

function WinAnsiToRawUnicode(const S: WinAnsiString): RawUnicode;
    Direct conversion of a WinAnsi (CodePage 1252) string into a Unicode encoded String
    - very fast, by using a fixed pre-calculated array for individual chars conversion

procedure WinAnsiToUnicodeBuffer(const S: WinAnsiString; Dest: PWordArray; DestLen: PtrInt);  
    Direct conversion of a WinAnsi (CodePage 1252) string into a Unicode buffer
    - very fast, by using a fixed pre-calculated array for individual chars conversion
    - text will be truncated if necessary to avoid buffer overflow in Dest[]

function WinAnsiToUtf8(WinAnsi: PAnsiChar; WinAnsiLen: PtrInt): RawUTF8; overload;
    Direct conversion of a WinAnsi (CodePage 1252) string into a UTF-8 encoded String
    - faster than SysUtils: don't use Utf8Encode(WideString) -> no Windows.Global(), and use a fixed pre-calculated array for individual chars conversion

function WinAnsiToUtf8(const S: WinAnsiString): RawUTF8; overload;
    Direct conversion of a WinAnsi (CodePage 1252) string into a UTF-8 encoded String
    - faster than SysUtils: don't use Utf8Encode(WideString) -> no Windows.Global(), and use a fixed pre-calculated array for individual chars conversion
**function** WordScanIndex(P: PWordArray; Count: PtrInt; Value: word): PtrInt;

*Fast search of an unsigned Word value position in a Word array*
- Count is the number of Word entries in P
- return index of P^[index]=Value
- return -1 if Value was not found

**function** WriteStringToStream(S: TStream; const Text: RawUTF8): boolean;

*Write an UTF-8 text into a TStream*
- format is Length(Integer):Text, i.e. the one used by ReadStringFromStream

**procedure** XorMemory(Dest,Source1,Source2: PByteArray; size: PtrInt); overload;

*Logical XOR of two memory buffers into a third*
- will perform on all buffer bytes:
  Dest[i] := Source1[i] xor Source2[i];

**procedure** XorMemory(Dest,Source: PByteArray; size: PtrInt); overload;

*Logical XOR of two memory buffers*
- will perform on all buffer bytes:
  Dest[i] := Dest[i] xor Source[i];

**function** xxHash32(crc: cardinal; P: PAnsiChar; len: integer): cardinal;

*Perform very fast xxHash hashing in 32-bit mode*
- will use optimized asm for x86/x64, or a pascal version on other CPUs

**procedure** YearToPChar(Y: PtrUInt; P: PUTF8Char);

*Add the 4 digits of integer Y to P^ as '0000'..'9999'*

**procedure** ZeroFill(Value: PVarData);

*Same as FillChar(Value^,SizeOf(TVarData),0)*
- so can be used for TVarData or Variant
- it will set V.VType := varEmpty, so Value will be Unassigned
- it won't call VarClear(variant(Value)): it should have been cleaned before

**function** _Arr(const Items: array of const; Options: TDocVariantOptions=[]): variant;

*Initialize a variant instance to store some document-based array content*
- array will be initialized with data supplied as parameters, e.g.
  aVariant := _Arr(["one",2,3.0]);
- this global function is an alias to TDocVariant.NewArray()
- by default, every internal value will be copied, so access of nested properties can be slow - if you expect the data to be read-only or not propagated into another place, set Options=[dvoValueCopiedByReference] or using _ArrFast() will increase the process speed a lot

**function** _ArrFast(const Items: array of const): variant; overload;

*Initialize a variant instance to store some document-based array content*
- this global function is an handy alias to:
  _Array(Items,JSON_OPTIONS[true]);
- so all created objects and arrays will be handled by reference, for best speed - but you should better write on the resulting variant tree with caution
procedure _ByRef(const DocVariant: variant; out Dest: variant; Options: TDocVariantOptions); overload;
  Copy a TDocVariant to another variable, changing the options on the fly
  - note that the content (items or properties) is copied by reference, so consider using _Copy()
  - instead if you expect to safely modify its content
  - will return null if the supplied variant is not a TDocVariant

function _ByRef(const DocVariant: variant; Options: TDocVariantOptions): variant;
  Copy a TDocVariant to another variable, changing the options on the fly
  - note that the content (items or properties) is copied by reference, so consider using _Copy()
  - instead if you expect to safely modify its content
  - will return null if the supplied variant is not a TDocVariant

function _Copy(const DocVariant: variant): variant;
  Return a full nested copy of a document-based variant instance
  - is just a wrapper around:
    TDocVariant.NewUnique(DocVariant,JSON_OPTIONS[false])
  - you can use this function to ensure that all internal properties of this variant will be copied
    per-value whatever options the nested objects or arrays were created with: to be used on a value
    returned as varByRef (e.g. by _() pseudo-method)
  - for huge document with a big depth of nested objects or arrays, a full per-value copy may be
    time and resource consuming, but will be also safe - consider using _ByRef() instead if a fast
    copy-by-reference is enough
  - will raise an EDocVariant if the supplied variant is not a TDocVariant or a varByRef pointing to a
    TDocVariant

function _CopyFast(const DocVariant: variant): variant;
  Return a full nested copy of a document-based variant instance
  - is just a wrapper around:
    TDocVariant.NewUnique(DocVariant,JSON_OPTIONS[true])
  - you can use this function to ensure that all internal properties of this variant will be copied
    per-value whatever options the nested objects or arrays were created with: to be used on a value
    returned as varByRef (e.g. by _() pseudo-method)
  - for huge document with a big depth of nested objects or arrays, a full per-value copy may be
    time and resource consuming, but will be also safe - consider using _ByRef() instead if a fast
    copy-by-reference is enough
  - will raise an EDocVariant if the supplied variant is not a TDocVariant or a varByRef pointing to a
    TDocVariant

function _CSV(const DocVariantOrString: variant): RawUTF8;
  Convert a TDocVariantData array or a string value into a CSV
  - will call either TDocVariantData.ToCSV, or return the string
  - returns "" if the supplied value is neither a TDocVariant or a string
  - could be used e.g. to store either a JSON CSV string or a JSON array of strings in a settings
    property
function _Json(const JSON: RawUTF8; var Value: variant; Options: TDocVariantOptions=[dvoReturnNullForUnknownProperty]): boolean; overload;

Initialize a variant instance to store some document-based content from a supplied (extended) JSON content
- this global function is an alias to TDocVariant.NewJSON(), and will return TRUE if JSON content was correctly converted into a variant
- in addition to the JSON RFC specification strict mode, this method will handle some BSON-like extensions, e.g. unquoted field names or ObjectID()
- by default, every internal value will be copied, so access of nested properties can be slow - if you expect the data to be read-only or not propagated into another place, add dvoValueCopiedByReference in Options will increase the process speed a lot, or use _JsonFast()

function _Json(const JSON: RawUTF8; Options: TDocVariantOptions=[dvoReturnNullForUnknownProperty]): variant; overload;

Initialize a variant instance to store some document-based content from a supplied (extended) JSON content
- this global function is an alias to TDocVariant.NewJSON(), and will return an Unassigned variant if JSON content was not correctly converted
- object or array will be initialized from the supplied JSON content, e.g.
  aVariant := _Json('{"id":10,"doc":{"name":"John","birthyear":1972}}');
  // now you can access to the properties via late binding
  assert(aVariant.id=10);
  assert(aVariant.doc.name='John');
  assert(aVariant.doc.birthYear=1972);
  // and also some pseudo-properties:
  assert(aVariant._count=2);
  assert(aVariant.doc._kind=ord(dvObject));
  aVariant := _Json('["one",2,3']);
  assert(aVariant._kind=ord(dvArray));
  for i := 0 to aVariant._count-1 do
    writeln(aVariant._(i));

- in addition to the JSON RFC specification strict mode, this method will handle some BSON-like extensions, e.g. unquoted field names:
  aVariant := _Json('{"id":10,"doc":{"name":"John","birthyear":1972}}');

- if the SynMongoDB unit is used in the application, the MongoDB Shell syntax will also be recognized to create TBSONVariant, like
  new Date() ObjectId() MinKey MaxKey /<jRegex>/<jOptions>
  see @http://docs.mongodb.org/manual/reference/mongodb-extended-json

- by default, every internal value will be copied, so access of nested properties can be slow - if you expect the data to be read-only or not propagated into another place, add dvoValueCopiedByReference in Options will increase the process speed a lot, or use _JsonFast()

function _JsonFast(const JSON: RawUTF8): variant;

Initialize a variant instance to store some document-based content from a supplied (extended) JSON content
- this global function is an handy alias to:
  _Json(JSON,JSON_OPTIONS[true]);

so it will return an Unassigned variant if JSON content was not correct
- so all created objects and arrays will be handled by reference, for best speed - but you should better write on the resulting variant tree with caution
- in addition to the JSON RFC specification strict mode, this method will handle some BSON-like extensions, e.g. unquoted field names or ObjectID()
function _JsonFastExt(const JSON: RawUTF8): variant;

Initialize a variant instance to store some extended document-based content
- this global function is an handy alias to:
  _Json(JSON,JSON_OPTIONS_FAST_EXTENDED);

function _JsonFastFmt(const Format: RawUTF8; const Args,Params: array of const): variant;

Initialize a variant instance to store some document-based content from a supplied (extended) JSON content, with parameters formatting
- this global function is an handy alias e.g. to:
  aVariant := _JsonFmt('{%:{$in:[?,?]}},{"type","food","snack"},JSON_OPTIONS[true]);
- so all created objects and arrays will be handled by reference, for best speed - but you should better write on the resulting variant tree with caution
- in addition to the JSON RFC specification strict mode, this method will handle some BSON-like extensions, e.g. unquoted field names or ObjectID():

procedure _JsonFmt(const Format: RawUTF8; const Args,Params: array of const; Options: TDocVariantOptions; out result: variant); overload;

Initialize a variant instance to store some document-based content from a supplied (extended) JSON content, with parameters formatting
- this overload function will set directly a local variant variable, and would be used by inlined _JsonFmt/_JsonFastFmt functions

function _JsonFmt(const Format: RawUTF8; const Args,Params: array of const; Options: TDocVariantOptions=[dvoReturnNullForUnknownProperty]): variant; overload;

Initialize a variant instance to store some document-based content from a supplied (extended) JSON content, with parameters formatting
- wrapper around the _Json(FormatUTF8(...,JSONFormat=true)) function, i.e. every Args[] will be inserted for each % and Params[] for each ?, with proper JSON escaping of string values, and writing nested _Obj() / _Arr() instances as expected JSON objects / arrays
- typical use (in the context of SynMongoDB unit) could be:
  aVariant := _JsonFmt('{%:{$in:[?,?]}},{"type","food","snack"}');
  aVariant := _JsonFmt('{type:{$in:?}}',[],[_Arr(["food","snack"])]);
  // which are the same as:
  aVariant := _JsonFmt('{type:{$in:["food","snack"]}}');
  // in this context:
  u := VariantSaveJSON(aVariant);
  assert(u='{"type":{"$in":["food","snack"]}}');
  u := VariantSaveMongoJSON(aVariant,modMongoShell);
  assert(u='{type:{$in:["food","snack"]}}');
- by default, every internal value will be copied, so access of nested properties can be slow - if you expect the data to be read-only or not propagated into another place, add dvoValueCopiedByReference in Options will increase the process speed a lot, or use _JsonFast()
function _Obj(const NameValuePairs: array of const; Options: TDocVariantOptions=[]): variant;

Initialize a variant instance to store some document-based object content
- object will be initialized with data supplied two by two, as Name,Value pairs, e.g.
  aVariant := _Obj(['name','John','year',1972]);

or even with nested objects:
  aVariant := _Obj(['name','John','doc',_Obj(['one',1,'two',2.0])]);

- this global function is an alias to TDocVariant.NewObject()
- by default, every internal value will be copied, so access of nested properties can be slow - if you expect the data to be read-only or not propagated into another place, set Options=[dvoValueCopiedByReference] or using _ObjFast() will increase the process speed a lot

procedure _ObjAddProps(const NameValuePairs: array of const; var Obj: variant); overload;

Add some property values to a document-based object content
- if Obj is a TDocVariant object, will add the Name/Value pairs
- if Obj is not a TDocVariant, will create a new fast document, initialized with supplied the Name/Value pairs
- this function will also ensure that ensure Obj is not stored by reference, but as a true TDocVariantData

procedure _ObjAddProps(const Document: variant; var Obj: variant); overload;

Add the property values of a document to a document-based object content
- if Document is not a TDocVariant object, will do nothing
- if Obj is a TDocVariant object, will add Document fields to its content
- if Obj is not a TDocVariant object, Document will be copied to Obj

function _ObjFast(const NameValuePairs: array of const): variant; overload;

Initialize a variant instance to store some document-based object content
- this global function is an handy alias to:
  Obj(NameValuePairs,JSON_OPTIONS[true]);
- so all created objects and arrays will be handled by reference, for best speed - but you should better write on the resulting variant tree with caution

function _ObjFast(aObject: TObject; aOptions: TTextWriterWriteObjectOptions=[woDontStoreDefault]): variant; overload;

Initialize a variant instance to store any object as a TDocVariant
- is a wrapper around _JsonFast(ObjectToJson(aObject,aOptions))

function _Safe(const DocVariant: variant; ExpectedKind: TDocVariantKind): PDocVariantData; overload;

Delphi has problems inlining this :( direct access to a TDocVariantData from a given variant instance
- return a pointer to the TDocVariantData corresponding to the variant instance, which may be of kind varByRef (e.g. when retrieved by late binding)
- will check the supplied document kind, i.e. either dvObject or dvArray and raise a EDocVariant exception if it does not match
function _Safe(const DocVariant: variant): PDocVariantData; overload;

Direct access to a TDocVariantData from a given variant instance
- return a pointer to the TDocVariantData corresponding to the variant instance, which may be of
  kind varByRef (e.g. when retrieved by late binding)
- will return a read-only fake TDocVariantData with Kind=dvUndefined if the supplied variant is not
  a TDocVariant instance, so could be safely used in a with block (use "with" moderation, of course):

```
with _Safe(aDocVariant)^ do
  for ndx := 0 to Count-1 do // here Count=0 for the "fake" result
    writeln(Names[ndx]);
```

or excluding the "with" statement, as more readable code:

```
var dv: PDocVariantData;
ndx: PtrInt;
begin
  dv := _Safe(aDocVariant);
  for ndx := 0 to dv.Count-1 do // here Count=0 for the "fake" result
    writeln(dv.Names[ndx]);
```

procedure _Unique(var DocVariant: variant);

Ensure a document-based variant instance will have only per-value nested objects or array
documents
- is just a wrapper around:
  TDocVariantData(DocVariant).InitCopy(DocVariant,JSON_OPTIONS[false])
- you can use this function to ensure that all internal properties of this variant will be copied
  per-value whatever options the nested objects or arrays were created with
- for huge document with a big depth of nested objects or arrays, a full per-value copy may be
time and resource consuming, but will be also safe
- will raise an EDocVariant if the supplied variant is not a TDocVariant or a varByRef pointing to a
  TDocVariant

procedure _UniqueFast(var DocVariant: variant);

Ensure a document-based variant instance will have only per-value nested objects or array
documents
- is just a wrapper around:
  TDocVariantData(DocVariant).InitCopy(DocVariant,JSON_OPTIONS[true])
- you can use this function to ensure that all internal properties of this variant will be copied
  per-reference whatever options the nested objects or arrays were created with
- for huge document with a big depth of nested objects or arrays, it will first create a whole copy
  of the document nodes, but further assignments of the resulting value will be per-reference, so
  will be almost instant
- will raise an EDocVariant if the supplied variant is not a TDocVariant or a varByRef pointing to a
  TDocVariant

Variables implemented in the SynCommons unit

AlgoSynLZ: TAlgoCompress;

Access to our fast SynLZ compression as a TAlgoCompress class
- please use this global variable methods instead of the deprecated
  SynLZCompress/SynLZDecompress wrapper functions

BiosInfoText: RawUTF8;

Some textual information about the current computer hardware, from BIOS
BOOL_UTF8: array[boolean] of RawUTF8;

JSON compatible representation of a boolean value, i.e. 'false' and 'true'
- can be used when a RawUTF8 string is expected

CacheResCount: integer = -1;

Current LoadResString() cached entries count
- i.e. resourcestring caching for faster use
- used only if a default system.pas is used, not our Extended version
- defined here, but resourcestring caching itself is implemented in the mORMoti18n.pas unit, if the ENHANCEDRTL conditional is not defined

CompareMemFixed: function(P1, P2: Pointer; Length: PtrInt): Boolean = CompareMem;

A CompareMem()-like function designed for small and fixed-sized content
- here, Length is expected to be a constant value - typically from sizeof() - so that inlining has better performance than calling the CompareMem() function

ConvertHexToBin: TNormTableByte;

A conversion table from hexa chars into binary data
- returns 255 for any character out of 0..9,A..Z,a..z range
- used e.g. by HexToBin() function
- is defined globally, since may be used from an inlined function

CpuFeatures: TIntelCpuFeatures;

The available CPU features, as recognized at program startup

CpuInfoText: RawUTF8;

Some textual information about the current CPU

crc32c: THasher;

Compute CRC32C checksum on the supplied buffer
- result is not compatible with zlib's crc32() - Intel/SCSI CRC32C is not the same polynom - but will use the fastest mean available, e.g. SSE 4.2, to achieve up to 16GB/s with the optimized implementation from SynCrypto.pas
- you should use this function instead of crc32cfast() or crc32csse42()

crc32cBy4: TCrc32cBy4;

Compute CRC32C checksum on one 32-bit unsigned integer
- can be used instead of crc32c() for inlined process during data acquisition
- doesn't make "crc := not crc" before and after the computation: caller has to start with "crc := cardinal(not 0)" and make "crc := not crc" at the end, to compute the very same hash value than regular crc32c()
- this variable will use the fastest mean available, e.g. SSE 4.2

crc32ctab: TCrc32ctab;

Tables used by crc32cfast() function
- created with a polynom diverse from zlib's crc32() algorithm, but compatible with SSE 4.2 crc32 instruction
- tables content is created from code in initialization section below
- will also be used internally by SymmetricEncrypt, FillRandom and TSynUniqueIdentifierGenerator as 1KB master/reference key tables
**CRC Blocks:**

```pascal
procedure crcblock(crc128, data128: PBlock128) = crcblockNoSSE42;
```

*Compute a proprietary 128-bit CRC of a 128-bit binary buffer*
- apply four crc32c() calls on the 128-bit input chunk, into a 128-bit crc
- its output won't match crc128c() value, which works on 8-bit input
- will use SSE 4.2 hardware accelerated instruction, if available
- is used e.g. by SynCrypto's TAESCFCRC to check for data integrity

```pascal
procedure crcblocks(crc128, data128: PBlock128; count: integer)=crcblocksfast;
```

*Compute a proprietary 128-bit CRC of 128-bit binary buffers*
- apply four crc32c() calls on the 128-bit input chunks, into a 128-bit crc
- its output won't match crc128c() value, which works on 8-bit input
- will use SSE 4.2 hardware accelerated instruction, if available
- is used e.g. by SynEcc's TECDHEProtocol.ComputeMAC for macCrc128c

**CurrentAnsiConvert:**

```pascal
TSynAnsiConvert;
```

*Global TSynAnsiConvert instance to handle current system encoding*
- this is the encoding as used by the AnsiString Delphi, so will be used before Delphi 2009 to speed-up VCL string handling (especially for UTF-8)
- this instance is global and instaniated during the whole program life time

**DefaultHasher:**

```pascal
THasher;
```

*The default hasher used by TDynArrayHashed*
- set to crc32cSse42() if SSE4.2 instructions are available on this CPU, or fallback to xxHash32() which performs better than crc32cfast()

**DefaultSynLogExceptionToStr:**

```pascal
TSynLogExceptionToStr = nil;
```

*Default exception logging callback - will be set by the SynLog unit*
- will add the default Exception details, including any Exception.Message
- if the exception inherits from ESynException
- returns TRUE: caller will then append ' at EAddr' and the stack trace

**DefaultTextWriterSerializer:**

```pascal
TTTextWriterClass = TTextWriterWithEcho;
```

*Contains the default JSON serialization class for WriteObject*
- if only SynCommons.pas is used, it will be TTextWriterWithEcho
- mORMot.pas will assign TJSONSerializer which uses RTTI to serialize TSQLRecord and any class published properties as JSON

**DocVariantType:**

```pascal
TDocVariant = nil;
```

*The internal custom variant type used to register TDocVariant*

**DocVariantVType:**

```pascal
integer = -1;
```

*Copy of DocVariantType.VarType*
- as used by inlined functions of TDocVariantData

**DOUBLE_PRECISION:**

```pascal
integer = 15;
```

*Best possible precision when rendering a "double" kind of float*
- can be used as parameter for ExtendedToShort/ExtendedToStr
- is defined as a var, so that you may be able to override the default settings, for the whole process

**DynArrayIsObjArray:**

```pascal
function(aDynArrayTypeInfo: Pointer): TPointerClassHashed;
```

*MORMot.pas will registry here its T*ObjArray serialization process*
- will be used by TDynArray.GetIsObjArray
DYNARRAY_HASHFIRSTFIELD: array[boolean,TDynArrayKind] of TDynArrayHashOne = ( (nil, HashByte, HashWord, HashInteger, HashInteger, HashInt64, HashInt64, HashInt64, HashInt64, HashInt64, HashInt64, HashInt64, HashAnsiString, HashAnsiString, HashAnsiString, HashAnsiString, HashWideString, HashSynUnicode, Hash128, Hash256, Hash512, HashPointer, HashVariant, nil), (nil, HashByte, HashByte, HashWord, HashInteger, HashInteger, HashInt64, HashInt64, HashInt64, HashInt64, HashInt64, HashInt64, HashAnsiStringI, HashAnsiStringI, HashAnsiStringI, HashAnsiStringI, HashWideStringI, HashSynUnicodeI, Hash128, Hash256, Hash512, HashPointer, HashVariantI, nil));

Helper array to get the hashing function corresponding to a given standard array type
- e.g. as DYNARRAY_HASHFIRSTFIELD[CaseInSensitive,djRawUTF8]
- not to be used as such, but e.g. when inlining TDynArray methods

DYNARRAY_SORTFIRSTFIELD: array[boolean,TDynArrayKind] of TDynArraySortCompare = ( (nil, SortDynArrayBoolean, SortDynArrayByte, SortDynArrayWord, SortDynArrayInteger, SortDynArrayCardinal, SortDynArraySingle, SortDynArrayInt64, SortDynArrayQWord, SortDynArrayDouble, SortDynArrayDouble, SortDynArrayAnsiString, SortDynArrayAnsiString, SortDynArrayAnsiString, SortDynArrayAnsiString, SortDynArrayUnicodeString, SortDynArray128, SortDynArray256, SortDynArray512, SortDynArrayPointer, SortDynArrayVariant, nil), (nil, SortDynArrayBoolean, SortDynArrayByte, SortDynArrayWord, SortDynArrayInteger, SortDynArrayCardinal, SortDynArraySingle, SortDynArrayInt64, SortDynArrayQWord, SortDynArrayDouble, SortDynArrayDouble, SortDynArrayAnsiString, SortDynArrayAnsiString, SortDynArrayAnsiString, SortDynArrayAnsiString, SortDynArrayUnicodeString, SortDynArray128, SortDynArray256, SortDynArray512, SortDynArrayPointer, SortDynArrayVariantI, nil));

Helper array to get the comparison function corresponding to a given standard array type
- e.g. as DYNARRAY_SORTFIRSTFIELD[CaseInSensitive,djRawUTF8]
- not to be used as such, but e.g. when inlining TDynArray methods

ExeVersion: TExeVersion;

Global information about the current executable and computer
- this structure is initialized in this unit's initialization block below
- you can call SetExecutableVersion() with a custom version, if needed

EXTENDED_PRECISION: integer = 18;
Best possible precision when rendering a "extended" kind of float
- can be used as parameter for ExtendedToShort/ExtendedToStr
- is defined as a var, so that you may be able to override the default settings, for the whole process

FillcharFast: procedure(var Dest; count: PtrInt; Value: byte);

Our fast version of FillChar()
- on Intel i386/x86_64, will use fast SSE2/ERMS instructions (if available), or optimized X87 assembly implementation for older CPUs
- on non-Intel CPUs, it will fallback to the default RTL FillChar()
- note: Delphi x86_64 is far from efficient: even ERMS was wrongly introduced in latest updates
GarbageCollector: TSynObjectList;

- A global "Garbage collector", for some classes instances which must live during whole main executable process
- used to avoid any memory leak with e.g. 'class var RecordProps', i.e. some singleton or static objects
- to be used, e.g. as:
  Version := TFileVersion.Create(InstanceFileName, DefaultVersion32);
  GarbageCollector.Add(Version);
- see also GarbageCollectorFreeAndNil() as an alternative

GarbageCollectorFreeing: boolean;

Set to TRUE when the global "Garbage collector" are being freed

GetBitsCountPtrInt: function(value: PtrInt): PtrInt = GetBitsCountPas;

- Compute how many bits are set in a given pointer-sized integer
- the PopCnt() intrinsic under FPC doesn't have any fallback on older CPUs, and default implementation is 5 times slower than our GetBitsCountPas() on x64
- this redirected function will use fast SSE4.2 popcnt opcode, if available

GetSystemTimePreciseAsFileTime: procedure(var ft: TFILETIME); stdcall;

- Returns the highest resolution possible UTC timestamp on this system
- detects newer API available since Windows 8, or fallback to good old GetSystemTimeAsFileTime() which may have the resolution of the HW timer, i.e. typically around 16 ms
- GetSystemTimeAsFileTime() is always faster, so is to be preferred if second resolution is enough (e.g. for UnixTimeUTC)
- see http://www.windowstimestamp.com/description

GetTickCount64: function: Int64; stdcall;

- The number of milliseconds that have elapsed since the system was started
- compatibility function, to be implemented according to the running OS
- will use the corresponding native API function under Vista+, or will emulate it for older Windows versions (XP)
- warning: FPC's SysUtils.GetTickCount64 or TThread.GetTickCount64 don't handle properly 49 days wrapping under XP -> always use this safe version

i18nDateText: function(const Iso: TTimeLog): string = nil;

- Custom TTimeLog date to ready to be displayed text function
- you can override this pointer in order to display the text according to your expected i18n settings
- this callback will therefore be set by the mORMoti18n.pas unit
- used e.g. by TTimeLogInt.18nText and by TSQLTable.ExpandAsString() methods, i.e. TSQLTableToGrid.DrawCell()

i18nDateTimeText: function(const DateTime: TDateTime): string = nil;

- Custom date to ready to be displayed text function
- you can override this pointer in order to display the text according to your expected i18n settings
- this callback will therefore be set by the mORMoti18n.pas unit
- used e.g. by TSQLTable.ExpandAsString() method, i.e. TSQLTableToGrid.DrawCell()

InterningHasher: THasher;

- The hash function used by TRawUTF8Interning
- set to crc32csse42() if SSE4.2 instructions are available on this CPU, or fallback to xxHash32() which performs better than crc32cfast()
IsWow64: boolean;

- Is set to TRUE if the current process is a 32-bit image running under WOW64
- WOW64 is the x86 emulator that allows 32-bit Windows-based applications to run seamlessly on 64-bit Windows
- equals always FALSE if the current executable is a 64-bit image

JSON_CHARS: TJsonCharSet;

- Branch-less table used for JSON parsing

JSON_CONTENT_TYPE_HEADER_VAR: RawUTF8;

- HTTP header for MIME content type used for plain JSON
- this global will be initialized with JSON_CONTENT_TYPE_HEADER constant, to avoid a memory allocation each time it is assigned to a variable

JSON_CONTENT_TYPE_VAR: RawUTF8;

- MIME content type used for JSON communication
- i.e. 'application/json; charset=UTF-8'
- this global will be initialized with JSON_CONTENT_TYPE constant, to avoid a memory allocation each time it is assigned to a variable

LoadResStringTranslate: procedure(var Text: string) = nil;

- These procedure type must be defined if a default system.pas is used
- mORMoti18n.pas unit will hack default LoadResString() procedure
- already defined in our Extended system.pas unit
- needed with FPC, Delphi 2009 and up, i.e. when ENHANCEDRTL is not defined
- expect generic "string" type, i.e. UnicodeString for Delphi 2009+
- not needed with the LVCL framework (we should be on server side)

MoveFast: procedure(const Source; var Dest; Count: PtrInt);

- Our fast version of move()
- on Delphi Intel i386/x86_64, will use fast SSE2 instructions (if available), or optimized X87 assembly implementation for older CPUs
- on non-Intel CPUs, it will fallback to the default RTL Move()

NormToLower: TNormTable;

- The NormToLower[] array is defined in our Enhanced RTL: define it now if it was not installed
- handle 8 bit upper chars as in WinAnsi / code page 1252 (e.g. accents)

NormToNorm: TNormTable;

- Case sensitive NormToUpper[]/NormToLower[]-like table
- i.e. NormToNorm[c] = c

NormToUpper: TNormTable;

- The NormToUpper[] array is defined in our Enhanced RTL: define it now if it was not installed
- handle 8 bit upper chars as in WinAnsi / code page 1252 (e.g. accents)

NormToUpperAnsi7: TNormTable;

- This table will convert 'a'..'z' into 'A'..'Z'
- so it will work with UTF-8 without decoding, whereas NormToUpper[] expects WinAnsi encoding

Null: variant absolute NullVarData;

- A slightly faster alternative to Variants.Null function
NULL_STR_VAR: RawUTF8;

Can be used to avoid a memory allocation for res := 'null'

OSVersion: TWindowsVersion;
The current Operating System version, as retrieved for the current process

OSVersion32: TOperatingSystemVersion;
The running Operating System

OSVersionInfo: TOSVersionInfoEx;
The current Operating System information, as retrieved for the current process

OSVersionInfoEx: RawUTF8;
Some additional system information as text, e.g. 'Wine 1.1.5'
- also always appended to OSVersionText high-level description

OSVersionText: RawUTF8;
The current Operating System version, as retrieved for the current process
- contains e.g. 'Windows Seven 64 SP1 (6.1.7601)' or 'Ubuntu 16.04.5 LTS - Linux 3.13.0 110
generic#157 Ubuntu SMP Mon Feb 20 11:55:25 UTC 2017'

OS_KIND: TOperatingSystem = osWindows;
The target Operating System used for compilation, as TOperatingSystem
- a specific Linux distribution may be detected instead of plain osLinux

PosExString: function(const SubStr, S: string; Offset: PtrUInt=1): PtrInt;
Our own PosEx() function dedicated to VCL string process
- Delphi XE or older don't support Pos() with an Offset

SetThreadNameInternal: procedure(ThreadID: TThreadID; const Name: RawUTF8) =
SetThreadNameDefault;
Is overridden e.g. by mORMot.pas to log the thread name

SINGLE_PRECISION: integer = 8;
Best possible precision when rendering a "single" kind of float
- can be used as parameter for ExtendedToShort/ExtendedToStr
- is defined as a var, so that you may be able to override the default settings, for the whole
process

SmallUInt32UTF8: array[0..999] of RawUTF8;
Naive but efficient cache to avoid string memory allocation for 0..999 small numbers by
Int32ToUTF8/UInt32ToUTF8
- use around 16KB of heap [since each item consumes 16 bytes], but increase overall performance
and reduce memory allocation (and fragmentation), especially during multi-threaded execution
- noticeable when strings are used as array indexes (e.g. in SynMongoDB BSON)
- is defined globally, since may be used from an inlined function

StrComp: function(Str1, Str2: pointer): PtrInt = StrCompFast;
Fastest available version of StrComp(), to be used with PUTF8Char/PAnsiChar
- won't use SSE4.2 instructions on supported CPUs by default, which may read some bytes beyond
the s string, so should be avoided e.g. over memory mapped files - call explicitly StrCompSSE42()
if you are confident on your input
**strcspn: function**(s, reject: pointer): integer = strcspnPas;

_Fastest available version of strcspn(), to be used with PUTF8Char/PAnsiChar_
- returns size of initial segment of s which doesn't appears in reject chars, e.g.
  `strcspn('1234,6789','1')=4`
- won't use SSE4.2 instructions on supported CPUs by default, which may read some bytes beyond the s string, so should be avoided e.g. over memory mapped files - call explicitly strcspnSSE42() if you are confident on your input

**StrLen: function**(S: pointer): PtrInt = StrLenPas;

_Our fast version of StrLen(), to be used with PUTF8Char/PAnsiChar_
- if available, a fast SSE2 asm will be used on Intel/AMD CPUs
- won't use SSE4.2 instructions on supported CPUs by default, which may read some bytes beyond the string, so should be avoided e.g. over memory mapped files - call explicitly StrLenSSE42() if you are confident on your input

**strspn: function**(s, accept: pointer): integer = strspnPas;

_Fastest available version of strspn(), to be used with PUTF8Char/PAnsiChar_
- returns size of initial segment of s which appears in accept chars, e.g.
  `strspn('abcdef','debca')=5`
- won't use SSE4.2 instructions on supported CPUs by default, which may read some bytes beyond the s string, so should be avoided e.g. over memory mapped files - call explicitly strspnSSE42() if you are confident on your input

SystemInfo: TSystemInfo;

_The current System information, as retrieved for the current process_
- under a WOW64 process, it will use the GetNativeSystemInfo() new API to retrieve the real top-most system information
- note that the lpMinimumApplicationAddress field is replaced by a more optimistic/realistic value ($100000 instead of default $10000)
- under BSD/Linux, only contain dwPageSize and dwNumberOfProcessors fields

TEXT_CHARS: TTextCharSet;

_Branch-less table used for text line/word/identifiers/uri parsing_

TSynLogExceptionToStrCustom: TSynLogExceptionToStr = nil;

_Allow to customize the ESynException logging message_

TwoDigitByteLookupW: packed[0..99] of word;

_Fast lookup table for converting any decimal number from 0 to 99 into their byte digits (0..9) equivalence_
- used e.g. by DoubleToAscii() implementing Grisu algorithm

TwoDigitLookupW: packed[0..99] of word absolute TwoDigitLookup;

_Fast lookup table for converting any decimal number from 0 to 99 into their ASCII ('0'..'9') equivalence_
UpperCopy255Buf: function(dest: PAnsiChar; source: PUTF8Char; sourceLen: PtrInt): PAnsiChar;

*Copy source^ into a 256 chars dest^ buffer with 7 bits upper case conversion*
- used internally for short keys match or case-insensitive hash
- returns final dest pointer
- will copy up to 255 AnsiChar (expect the dest buffer to be defined e.g. as array[byte] of AnsiChar on the caller stack)
- won’t use SSE4.2 instructions on supported CPUs by default, which may read some bytes beyond the s string, so should be avoided e.g. over memory mapped files - call explicitly
  UpperCopy255BufSSE42() if you are confident on your input

UTF8AnsiConvert: TSynAnsiUTF8;

*Global TSynAnsiConvert instance to handle UTF-8 encoding (code page CP_UTF8)*
- this instance is global and instantiated during the whole program lifetime

WinAnsiConvert: TSynAnsiFixedWidth;

*Global TSynAnsiConvert instance to handle WinAnsi encoding (code page 1252)*
- this instance is global and instantiated during the whole program lifetime
- it will be created from hard-coded values, and not using the system API, since it appeared that some systems (e.g. in Russia) did tweak the registry so that 1252 code page maps 1251 code page
27.6. SynCrtSock.pas unit

_Purpose_: Classes implementing TCP/UDP/HTTP client and server protocol
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The _SynCrtSock_ unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.1.1.2.4</td>
<td>HTTP/1.1 protocol</td>
<td>2545</td>
</tr>
</tbody>
</table>

Units used in the _SynCrtSock_ unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynWinSock</td>
<td>Low level access to network Sockets for the Win32 platform - this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1843</td>
</tr>
</tbody>
</table>

_SynCrtSock class hierarchy_

Objects implemented in the _SynCrtSock_ unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>ECrtSocket</td>
<td>Exception thrown by the classes of this unit</td>
<td>1085</td>
</tr>
<tr>
<td>EWinHTTP</td>
<td>WinHTTP exception type</td>
<td>1118</td>
</tr>
<tr>
<td>EWinINet</td>
<td>WinINet exception type</td>
<td>1118</td>
</tr>
<tr>
<td>TCrtSocket</td>
<td>Fast low-level Socket implementation</td>
<td>1086</td>
</tr>
<tr>
<td>THttpApiServer</td>
<td>HTTP server using fast http.sys kernel-mode server</td>
<td>1103</td>
</tr>
<tr>
<td>THttpApiWebSocketConnection</td>
<td>Structure representing a single WebSocket connection</td>
<td>1106</td>
</tr>
<tr>
<td>THttpApiWebSocketServer</td>
<td>HTTP &amp; WebSocket server using fast http.sys kernel-mode server</td>
<td>1108</td>
</tr>
<tr>
<td>THttpApiWebSocketServerProto</td>
<td>Protocol Handler of websocket endpoints events</td>
<td>1107</td>
</tr>
<tr>
<td>THttpClientSocket</td>
<td>Socket API based REST and HTTP/1.1 compatible client class</td>
<td>1093</td>
</tr>
<tr>
<td>THttpRequest</td>
<td>To have existing RTTI for published properties abstract class to handle HTTP/1.1 request</td>
<td>1114</td>
</tr>
<tr>
<td>THttpRequestExtendedOptions</td>
<td>A record to set some extended options for HTTP clients</td>
<td>1114</td>
</tr>
<tr>
<td>THttpServer</td>
<td>Main HTTP server Thread using the standard Sockets API (e.g. WinSock)</td>
<td>1110</td>
</tr>
<tr>
<td>THttpServerGeneric</td>
<td>Abstract class to implement a HTTP server</td>
<td>1099</td>
</tr>
<tr>
<td>THttpServerRequest</td>
<td>A generic input/output structure used for HTTP server requests</td>
<td>1097</td>
</tr>
<tr>
<td>THttpServerResp</td>
<td>HTTP response Thread as used by THttpServer Socket API based class</td>
<td>1095</td>
</tr>
<tr>
<td>THttpServerSocket</td>
<td>Socket API based HTTP/1.1 server class used by THttpServer Threads</td>
<td>1093</td>
</tr>
<tr>
<td>THttpSocket</td>
<td>Parent of THttpClientSocket and THttpServerSocket classes</td>
<td>1091</td>
</tr>
<tr>
<td>THttpSocketCompressRecord</td>
<td>Used to maintain a list of known compression algorithms</td>
<td>1091</td>
</tr>
<tr>
<td>TMacAddress</td>
<td>Interface name/address pairs as returned by GetMacAddresses</td>
<td>1120</td>
</tr>
<tr>
<td>TPollAsynchSockets</td>
<td>Read/write buffer-oriented process of multiple non-blocking connections</td>
<td>1124</td>
</tr>
<tr>
<td>TPollSocketAbstract</td>
<td>Abstract parent class for efficient socket polling</td>
<td>1120</td>
</tr>
<tr>
<td>TPollSocketPoll</td>
<td>Socket polling via poll/WSAPoll API</td>
<td>1122</td>
</tr>
<tr>
<td>TPollSocketResult</td>
<td>Modifications notified by TPollSocketAbstract.WaitForModified</td>
<td>1120</td>
</tr>
<tr>
<td>TPollSockets</td>
<td>Implements efficient polling of multiple sockets</td>
<td>1122</td>
</tr>
<tr>
<td>TPollSocketSelect</td>
<td>Socket polling via Windows' Select() API</td>
<td>1121</td>
</tr>
<tr>
<td>TPollSocketsSlot</td>
<td>Store information of one TPollAsynchSockets connection</td>
<td>1124</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>TServerGeneric</td>
<td>Abstract class to implement a server thread</td>
<td>1099</td>
</tr>
<tr>
<td>TSimpleHttpClient</td>
<td>Simple wrapper around THttpClientSocket/THttpRequest instances</td>
<td>1119</td>
</tr>
<tr>
<td>TSMTPConnection</td>
<td>May be used to store a connection to a SMTP server</td>
<td>1120</td>
</tr>
<tr>
<td>TSynThread</td>
<td>A simple TThread with a &quot;Terminate&quot; event run in the thread context</td>
<td>1095</td>
</tr>
<tr>
<td>TSynThreadPool</td>
<td>A simple Thread Pool, used e.g. for fast handling HTTP requests</td>
<td>1096</td>
</tr>
<tr>
<td>TSynThreadPoolHttpApiWebSocketServer</td>
<td>A Thread Pool, used for fast handling WebSocket requests</td>
<td>1109</td>
</tr>
<tr>
<td>TSynThreadPoolSubThread</td>
<td>I/O completion ports API is the best option under Windows under Linux/POSIX, we fallback to a classical event-driven pool defines the sub-threads used by TSynThreadPool</td>
<td>1096</td>
</tr>
<tr>
<td>TSynThreadPoolTHttpServer</td>
<td>A simple Thread Pool, used for fast handling HTTP requests of a THttpServer</td>
<td>1097</td>
</tr>
<tr>
<td>TSynWebSocketGuard</td>
<td>Thread for closing WebSocket connections which not response more than PingTimeout interval</td>
<td>1109</td>
</tr>
<tr>
<td>TURI</td>
<td>Structure used to parse an URI into its components</td>
<td>1113</td>
</tr>
<tr>
<td>TWinHTTP</td>
<td>A class to handle HTTP/1.1 request using the WinHTTP API</td>
<td>1118</td>
</tr>
<tr>
<td>TWinHttpAPI</td>
<td>A class to handle HTTP/1.1 request using either WinINet or WinHTTP API</td>
<td>1117</td>
</tr>
<tr>
<td>TWinHTTPUpgradeable</td>
<td>A class to establish a client connection to a WebSocket server using Windows API</td>
<td>1118</td>
</tr>
<tr>
<td>TWinHTTPWebSocketClient</td>
<td>WebSocket client implementation</td>
<td>1118</td>
</tr>
<tr>
<td>TWinINet</td>
<td>A class to handle HTTP/1.1 request using the WinINet API</td>
<td>1117</td>
</tr>
</tbody>
</table>

**ECrtSocket = class(Exception)**

*Exception thrown by the classes of this unit*

**constructor Create(const Msg: string; Error: integer); overload;**
Will concat the message with the supplied WSAGetLastError information

**constructor Create(const Msg: string); overload;**
Will concat the message with the WSAGetLastError information

**constructor CreateFmt(const Msg: string; const Args: array of const; Error: integer); overload;**
Will concat the message with the supplied WSAGetLastError information

**property LastError: integer read fLastError;**
The associated WSAGetLastError value
**TCrtSocket = class(TObject)**

*Fast low-level Socket implementation*
- direct access to the OS (Windows, Linux) network layer API
- use Open constructor to create a client to be connected to a server
- use Bind constructor to initialize a server
- use SockIn and SockOut (after CreateSock*) to read/readln or write/writeln as with standard Delphi text files (see SendEmail implementation)
- even if you do not use read(SockIn^), you may call CreateSockIn then read the (binary) content via SockInRead/SockInPending methods, which would benefit of the SockIn^ input buffer to maximize reading speed
- to write data, CreateSockOut and write(SockOut^) is not mandatory: you rather may use SockSend() overloaded methods, followed by a SockFlush call
- in fact, you can decide whatever to use none, one or both SockIn/SockOut
- since this class rely on its internal optimized buffering system, TCP_NODELAY is set to disable the Nagle algorithm
- our classes are (much) faster than the Indy or Synapse implementation

**constructor** Bind(**const** aAddr: SockString; aLayer: TCrtSocketLayer=cslTCP; aTimeOut: integer=10000);

*Bind to an address*
- aAddr='1234' - bind to a port on all interfaces, the same as '0.0.0.0:1234'
- aAddr='IP:port' - bind to specified interface only, e.g. '1.2.3.4:1234'
- aAddr='unix:/path/to/file' - bind to unix domain socket, e.g. 'unix:/run/mormot.sock'
- aAddr='' - bind to systemd descriptor on linux. See http://0pointer.de/blog/projects/socket-activation.html

**constructor** Create(aTimeOut: PTrInt=10000); **reintroduce; virtual;**

*Common initialization of all constructors*
- do not call directly, but use Open / Bind constructors instead

**constructor** Open(**const** aServer, aPort: SockString; aLayer: TCrtSocketLayer=cslTCP; aTimeOut: cardinal=10000; aTLS: boolean=false);

*Connect to aServer:aPort*
- you may ask for a TLS secured client connection (only available under Windows by now, using the SChannel API)

**destructor** Destroy; **override;**

*Close the opened socket, and corresponding SockIn/SockOut*

**function** AcceptIncoming(ResultClass: TCrtSocketClass=nil): TCrtSocket;

*Direct accept an new incoming connection on a bound socket*
- instance should have been setup as a server via a previous Bind() call
- returns nil on error or a ResultClass instance on success
- if ResultClass is nil, will return a plain TCrtSocket, but you may specify e.g. THttpServerSocket if you expect incoming HTTP requests

**function** LastLowSocketError: Integer;

*Returns the low-level error number*
- i.e. returns WSAGetLastError
function PeerAddress: SockString;
    *Remote IP address of the last packet received (SocketLayer=slUDP only)*

function PeerPort: integer;
    *Remote IP port of the last packet received (SocketLayer=slUDP only)*

function SockConnected: boolean;
    *Check the connection status of the socket*

function SockInPending(aTimeOutMS: integer; aPendingAlsoInSocket: boolean=false): integer;
    *Returns the number of bytes in SockIn buffer or pending in Sock*
    - if SockIn is available, it first check from any data in SockIn^.Buffer, then call InputSock to try to receive any pending data if the buffer is void
    - if aPendingAlsoInSocket is TRUE, returns the bytes available in both the buffer and the socket (sometimes needed, e.g. to process a whole block at once)
    - will wait up to the specified aTimeOutMS value (in milliseconds) for incoming data - may wait a little less time on Windows due to a select bug
    - returns -1 in case of a socket error (e.g. broken/closed connection); you can raise a ECrtSocket exception to propagate the error

function SockInRead(Content: PAnsiChar; Length: integer; UseOnlySockIn: boolean=false): integer;
    *Read Length bytes from SockIn buffer + Sock if necessary*
    - if SockIn is available, it first gets data from SockIn^.Buffer, then directly receive data from socket if UseOnlySockIn=false
    - if UseOnlySockIn=true, it will return the data available in SockIn^, and returns the number of bytes
    - can be used also without SockIn: it will call directly SockRecv() in such case (assuming UseOnlySockIn=false)

function SockReceivePending(TimeOutMS: integer): TCrtSocketPending;
    *Check if there are some pending bytes in the input sockets API buffer*
    - returns cspSocketError if the connection is broken or closed
    - warning: on Windows, may wait a little less than TimeOutMS (select bug)

function SockReceiveString: SockString;
    *Returns the socket input stream as a string*

function SockSendRemainingSize: integer;
    *How many bytes could be added by SockSend() in the internal buffer*

function TrySndLow(P: pointer; Len: integer): boolean;
    *Direct send data through network*
    - return false on any error, true on success
    - bypass the SndBuf or SockOut^ buffers
function TrySockRecv(Buffer: pointer; var Length: integer; StopBeforeLength: boolean=false): boolean;

*Fill the Buffer with Length bytes*
- use TimeOut milliseconds wait for incoming data
- bypass the SockIn^ buffers
- return false on any fatal socket error, true on success
- call Close if the socket is identified as shutdown from the other side
- you may optionally set StopBeforeLength=true, then the read bytes count are set in Length, even if not all expected data has been received - in this case, Close method won't be called

function TrySockSendFlush: boolean;

*Flush all pending data to be sent*
- returning true on success

procedure AcceptRequest(aClientSock: TSocket; aClientSin: PVarSin);

*Initialize the instance with the supplied accepted socket*
- is called from a bound TCP Server, just after Accept()

procedure Close;

*Close and shutdown the connection (called from Destroy)*

procedure CloseSockIn;

*Finalize SockIn receiving buffer*
- you may call this method when you are sure that you don't need the input buffering feature on this connection any more (e.g. after having parsed the HTTP header, then rely on direct socket communication)

procedure CloseSockOut;

*Finalize SockOut receiving buffer*
- you may call this method when you are sure that you don't need the output buffering feature on this connection any more (e.g. after having parsed the HTTP header, then rely on direct socket communication)

procedure CreateSockIn(LineBreak: TTextLineBreakStyle=tlbsCRLF; InputBufferSize: Integer=1024);

*Initialize SockIn for receiving with read[ln](SockIn^,...)*
- data is buffered, filled as the data is available
- read(char) or readln() is indeed very fast
- multithread applications would also use this SockIn pseudo-text file
- by default, expect CR+LF as line feed (i.e. the HTTP way)

procedure CreateSockOut(OutputBufferSize: Integer=1024);

*Initialize SockOut for sending with write[ln](SockOut^,....)*
- data is sent (flushed) after each writeln() - it's a compiler feature
- use rather SockSend() + SockSendFlush to send headers at once e.g. since writeln(SockOut^,..) flush buffer each time

procedure OpenBind(const aServer, aPort: SockString; doBind: boolean; aSock: integer=-1; aLayer: TCrtSocketLayer=cslTCP; aTLS: boolean=false);

*Low-level internal method called by Open() and Bind() constructors*
- raise an ECrtSocket exception on error
- you may ask for a TLS secured client connection (only available under Windows by now, using the SChannel API)
procedure SndLow(P: pointer; Len: integer);

Direct send data through network
- raise a ECrtSocket exception on any error
- bypass the SockSend() or SockOut^ buffers

procedure SockRecv(Buffer: pointer; Length: integer);

Fill the Buffer with Length bytes
- use TimeOut milliseconds wait for incoming data
- bypass the SockIn^ buffers
- raise ECrtSocket exception on socket error

procedure SockRecvLn(out Line: SockString; CROnly: boolean=false); overload;

Call readln(SockIn^,Line) or simulate it with direct use of Recv(Sock, ..)
- char are read one by one if needed
- use TimeOut milliseconds wait for incoming data
- raise ECrtSocket exception on socket error
- by default, will handle #10 or #13#10 as line delimiter (as normal text files), but you can delimit lines using #13 if CROnly is TRUE

procedure SockRecvLn; overload;

Call readln(SockIn^) or simulate it with direct use of Recv(Sock, ..)
- char are read one by one
- use TimeOut milliseconds wait for incoming data
- raise ECrtSocket exception on socket error
- line content is ignored

procedure SockSend(const Line: SockString=''); overload;

Simulate writeln() with a single line - includes trailing #13#10

procedure SockSend(const Values: array of const); overload;

Simulate writeln() with direct use of Send(Sock, ..) - includes trailing #13#10
- useful on multi-threaded environment (as in THttpServer.Process)
- no temp buffer is used
- handle SockString, ShortString, Char, Integer parameters
- raise ECrtSocket exception on socket error

procedure SockSend(P: pointer; Len: integer); overload;

Append P^ data into SndBuf (used by SockSend(), e.g.) - no trailing #13#10
- call SockSendFlush to send it through the network via SndLow()

procedure SockSendFlush(const aBody: SockString=''); virtual;

Flush all pending data to be sent, optionally with some body content
- raise ECrtSocket on error

procedure Write(const Data: SockString);

Direct send data through network
- raise a ECrtSocket exception on any error
- bypass the SndBuf or SockOut^ buffers
- raw Data is sent directly to OS: no LF/CRLF is appended to the block

property BytesIn: Int64 read fBytesIn;

Total bytes received
property BytesOut: Int64 read fBytesOut;
    Total bytes sent

property KeepAlive: Integer index SO_KEEPALIVE write SetInt32OptionByIndex;
    Set the SO_KEEPALIVE option for the connection
    - 1 (true) will enable keep-alive packets for the connection

property Linger: Integer index SO_LINGER write SetInt32OptionByIndex;
    Set the SO_LINGER option for the connection, to control its shutdown
    - by default (or Linger<0), Close will return immediately to the caller, and any pending data will
      be delivered if possible
    - Linger > 0 represents the time in seconds for the timeout period to be applied at Close; under
      Linux, will also set SO_REUSEADDR; under Darwin, set SO_NOSIGPIPE
    - Linger = 0 causes the connection to be aborted and any pending data is immediately discarded
      at Close

property Port: SockString read fPort;
    IP port, initialized after Open() with port number

property ReceiveTimeout: Integer index SO_RCVTIMEO write SetInt32OptionByIndex;
    Set the SO_RCVTIMEO option for the connection
    - i.e. the timeout, in milliseconds, for blocking receive calls
    - see http://msdn.microsoft.com/en-us/library/windows/desktop/ms740476

property RemoteIP: SockString read fRemoteIP write fRemoteIP;
    Remote IP address after AcceptRequest() call over TCP
    - is either the raw connection IP to the current server socket, or a custom header value set by a
      local proxy as retrieved by inherited THttpServerSocket.GetRequest, searching the header
      named in THttpServerGeneric.RemoteIPHeader (e.g. 'X-Real-IP' for nginx)

property SendTimeout: Integer index SO_SNDTIMEO write SetInt32OptionByIndex;
    Set the SO_SNDTIMEO option for the connection
    - i.e. the timeout, in milliseconds, for blocking send calls
    - see http://msdn.microsoft.com/en-us/library/windows/desktop/ms740476

property Server: SockString read fServer;
    IP address, initialized after Open() with Server name

property Sock: TSocket read fSock write fSock;
    Low-level socket handle, initialized after Open() with socket

property SocketLayer: TCrtSocketLayer read fSocketLayer;
    Low-level socket type, initialized after Open() with socket

property SockIn: PTextFile read fSockIn;
    After CreateSockIn, use Readln(SockIn^,s) to read a line from the opened socket

property SockOut: PTextFile read fSockOut;
    After CreateSockOut, use Writeln(SockOut^,s) to send a line to the opened socket
property TCPNoDelay: Integer index TCP_NODELAY write SetInt32OptionByIndex;
Set the TCP_NODELAY option for the connection
- default 1 (true) will disable the Nagle buffering algorithm; it should only be set for applications
  that send frequent small bursts of information without getting an immediate response, where
timely delivery of data is required - so it expects buffering before calling Write() or SndLow()
- you can set 0 (false) here to enable the Nagle algorithm, if needed
  - see http://www.unixguide.net/network/socketfaq/2.16.shtml

property TimeOut: PtrInt read fTimeOut;
If higher than 0, read loop will wait for incoming data till TimeOut milliseconds (default value is
10000) - used also in SockSend()

THttpSocketCompressRec = record
Used to maintain a list of known compression algorithms
CompressMinSize: integer;
The size in bytes after which compress will take place
- will be 1024 e.g. for 'zip' or 'deflate'
- could be 0 e.g. when encrypting the content, meaning "always compress"
Func: THttpSocketCompress;
The function handling compression and decompression
Name: SockString;
The compression name, as in ACCEPT-ENCODING: header (gzip, deflate, synlzo)

THttpSocket = class(TCrtSocket)
Parent of THttpClientSocket and THttpServerSocket classes
- contain properties for implementing HTTP/1.1 using the Socket API
- handle chunking of body content
- can optionally compress and uncompress on the fly the data, with standard gzip/deflate or
  custom (synlzo/synlz) protocols
Command: SockString;
Will contain the first header line:
- 'GET /path HTTP/1.1' for a GET request with THttpServer, e.g.
- 'HTTP/1.0 200 OK' for a GET response after Get() e.g.

Content: SockString;
Will contain the data retrieved from the server, after the Request

ContentLength: integer;
Same as HeaderGetValue('CONTENT-LENGTH'), but retrieved during Request
- is overridden with real Content length during HTTP body retrieval

ContentType: SockString;
Same as HeaderGetValue('CONTENT-TYPE'), but retrieved during Request

HeaderFlags: set of(transferChunked, connectionClose, connectionUpgrade, connectionKeepAlive, hasRemoteIP);
Map the presence of some HTTP headers, but retrieved during Request
Headers: SockString;

- Will contain all header lines after a Request
  - use HeaderGetValue() to get one HTTP header item value by name

ServerInternalState: integer;

- Same as HeaderGetValue('SERVER-INTERNALSTATE'), but retrieved during Request
  - proprietary header, used with our RESTful ORM access

TCP/IP prefix to mask HTTP protocol

- if not set, will create full HTTP/1.0 or HTTP/1.1 compliant content
- in order to make the TCP/IP stream not HTTP compliant, you can specify a prefix which will be put before the first header line: in this case, the TCP/IP stream won't be recognized as HTTP, and will be ignored by most AntiVirus programs, and increase security - but you won't be able to use an Internet Browser nor AJAX application for remote access any more

Upgrade: SockString;

- Same as HeaderGetValue('UPGRADE'), but retrieved during Request

XPoweredBy: SockString;

- Same as HeaderGetValue('X-POWERED-BY'), but retrieved during Request

function HeaderGetText(const aRemoteIP: SockString=''): SockString;

- Get all Header values at once, as CRLF delimited text
  - you can optionally specify a value to be added as 'RemoteIP: ' header

function HeaderGetValue(const aUpperName: SockString): SockString;

- HeaderGetValue('CONTENT-TYPE')='text/html', e.g.
  - supplied aUpperName should be already uppecased

function RegisterCompress(aFunction: THttpGetSocketCompress; aCompressMinSize: integer=1024): boolean;

- Will register a compression algorithm
  - used e.g. to compress on the fly the data, with standard gzip/deflate or custom (synlzo/synlz) protocols
  - returns true on success, false if this function or this ACCEPT-ENCODING: header was already registered
  - you can specify a minimal size (in bytes) before which the content won't be compressed (1024 by default, corresponding to a MTU of 1500 bytes)
  - the first registered algorithm will be the prefered one for compression

procedure GetBody;

- Retrieve the HTTP body (after uncompression if necessary) into Content

procedure GetHeader(HeadersUnFiltered: boolean=false);

- Retrieve the HTTP headers into Headers[] and fill most properties below
  - only relevant headers are retrieved, unless HeadersUnFiltered is set

procedure HeaderAdd(const aValue: SockString);

- Add an header 'name: value' entry

procedure HeaderSetText(const aText: SockString; const aForcedContentType: SockString='');

- Set all Header values at once, from CRLF delimited text
THttpServerSocket = class(THttpSocket)
Socket API based HTTP/1.1 server class used by THttpServer Threads

constructor Create(aServer: THttpServer); reintroduce;
Create the socket according to a server
- will register the THttpSocketCompress functions from the server
- once created, caller should call AcceptRequest() to accept the socket

function GetRequest(withBody: boolean; headerMaxTix: Int64): THttpServerSocketGetRequestResult; virtual;
Main object function called after aClientSock := Accept + Create:
- get Command, Method, URL, Headers and Body (if withBody is TRUE)
- get sent data in Content (if withBody=true and ContentLength<>0)
- returned enumeration will indicates the processing state

property KeepAliveClient: boolean read fKeepAliveClient write fKeepAliveClient;
True if the client is HTTP/1.1 and 'Connection: Close' is not set
- default HTTP/1.1 behavior is "keep alive", unless 'Connection: Close' is specified, cf. RFC 2068 page 108: "HTTP/1.1 applications that do not support persistent connections MUST include the "close" connection option in every message"

property Method: SockString read fMethod;
Contains the method ('GET','POST'.. e.g.) after GetRequest()

property RemoteConnectionID: THttpServerConnectionID read fRemoteConnectionID;
The recognized connection ID, after a call to GetRequest()
- identifies either the raw connection on the current server, or is a custom header value set by a local proxy, e.g. THttpServerGeneric.RemoteConnIDHeader='X-Conn-ID' for nginx

property URL: SockString read fURL;
Contains the URL ('/' e.g.) after GetRequest()

THttpClientSocket = class(THttpSocket)
Socket API based REST and HTTP/1.1 compatible client class
- this component is HTTP/1.1 compatible, according to RFC 2068 document
- the REST commands (GET/POST/PUT/DELETE) are directly available
- open connection with the server with inherited Open(server,port) function
- if KeepAlive>0, the connection is not broken: a further request (within KeepAlive milliseconds) will use the existing connection if available, or recreate a new one if the former is outdated or reset by server (will retry only once); this is faster, uses less resources (especially under Windows), and is the recommended way to implement a HTTP/1.1 server
- on any error (timeout, connection closed) will retry once to get the value
- don't forget to use Free procedure when you are finished
constructor Create(aTimeOut: PInt=0); override;

Common initialization of all constructors
- this overridden method will set the UserAgent with some default value
- you can customize the default client timeouts by setting appropriate aTimeout parameters (in ms) if you left the 0 default parameters, it would use global HTTP_DEFAULT_RECEIVETIMEOUT variable values

function Delete(const url: SockString; KeepAlive: cardinal=0; const header: SockString=''): integer;

After an Open(server,port), return 200,202,204 if OK, http status error otherwise

function Get(const url: SockString; KeepAlive: cardinal=0; const header: SockString=''): integer;

After an Open(server,port), return 200 if OK, http status error otherwise
- get the page data in Content

function GetAuth(const url, AuthToken: SockString; KeepAlive: cardinal=0): integer;

After an Open(server,port), return 200 if OK, http status error otherwise
- get the page data in Content
- if AuthToken='', will add an header with 'Authorization: Bearer ' +AuthToken

function Head(const url: SockString; KeepAlive: cardinal=0; const header: SockString=''): integer;

After an Open(server,port), return 200 if OK, http status error otherwise - only header is read from server: Content is always '', but Headers are set

function Post(const url, Data, DataType: SockString; KeepAlive: cardinal=0; const header: SockString=''): integer;

After an Open(server,port), return 200,201,204 if OK, http status error otherwise

function Put(const url, Data, DataType: SockString; KeepAlive: cardinal=0; const header: SockString=''): integer;

After an Open(server,port), return 200,201,204 if OK, http status error otherwise

function Request(const url, method: SockString; KeepAlive: cardinal; const header, Data, DataType: SockString; retry: boolean): integer; virtual;

Low-level HTTP/1.1 request
- called by all Get/Head/Post/Put/Delete REST methods
- after an Open(server,port), return 200,202,204 if OK, http status error otherwise
- retry is false by caller, and will be recursively called with true to retry once

property ProcessName: SockString read fProcessName write fProcessName;
The associated process name

property UserAgent: SockString read fUserAgent write fUserAgent;

By default, the client is identified as IE 5.5, which is very friendly welcome by most servers :(
- you can specify a custom value here
TSynThread = class(TThread)

A simple TThread with a "Terminate" event run in the thread context
- the TThread.OnTerminate event is run within Synchronize() so did not match our expectations to be able to release the resources in the thread context which created them (e.g. for COM objects, or some DB drivers)
- used internally by THttpServerGeneric.NotifyThreadStart() - you should not have to use the protected fOnThreadTerminate event handler
- also define a Start method for compatibility with older versions of Delphi

constructor Create(CreateSuspended: boolean); reintroduce; virtual;
  Initialize the server instance, in non suspended state

function SleepOrTerminated(MS: cardinal): boolean;
  Safe version of Sleep() which won't break the thread process
  - returns TRUE if the thread was Terminated
  - returns FALSE if successfully waited up to MS milliseconds

procedure Start;
  Method to be called when the thread was created as suspended
  - Resume is deprecated in the newest RTL, since some OS - e.g. Linux - do not implement this pause/resume feature
  - we define here this method for older versions of Delphi

property Terminated;
  Defined as public since may be used to terminate the processing methods

THttpServerResp = class(TSynThread)

HTTP response Thread as used by THttpServer Socket API based class
- Execute procedure get the request and calculate the answer, using the thread for a single client connection, until it is closed
- you don't have to overload the protected THttpServerResp Execute method: override THttpServer.Request() function or, if you need a lower-level access (change the protocol, e.g.) THttpServer.Process() method itself

constructor Create(aServerSock: THttpServerSocket; aServer: THttpServer);
  Initialize the response thread for the corresponding incoming socket
  - this version will handle KeepAlive, for such an incoming request
reintroduce; overload; virtual;

constructor Create(aSock: TSocket; const aSin: TVarSin; aServer: THttpServer);
  Initialize the response thread for the corresponding incoming socket
  - this version will get the request directly from an incoming socket
reintroduce; overload;

property ConnectionID: THttpServerConnectionID read fConnectionID;
  The unique identifier of this connection

property Server: THttpServer read fServer;
  The associated main HTTP server instance
property ServerSock: THttpServerSocket read fServerSock;
  The associated socket to communicate with the client

TSynThreadPoolSubThread = class(TSynThread)
  I/O completion ports API is the best option under Windows under Linux/POSIX, we fallback to a
classical event-driven pool defines the sub-threads used by TSynThreadPool

  constructor Create(Owner: TSynThreadPool); reintroduce;
    Exception-safe call of fOwner.Task() initialize the thread

  destructor Destroy; override;
    Finalize the thread

  procedure Execute; override;
    Will loop for any pending task, and execute fOwner.Task()

TSynThreadPool = class(TObject)
  A simple Thread Pool, used e.g. for fast handling HTTP requests
  - implemented over I/O Completion Ports under Windows, or a classical Event-driven approach
    under Linux/POSIX

  constructor Create(NumberOfThreads: Integer=32; aOverlapHandle: THandle=INVALID_HANDLE_VALUE );
    initialize a thread pool with the supplied number of threads
    - up to 256 threads can be associated to a Thread Pool
    - can optionally accept aOverlapHandle - a handle previously opened for overlapped I/O (IOCP)
      under Windows
    - aQueuePendingContext=true will store the pending context into an internal queue, so that
      Push() always returns true

  destructor Destroy; override;
    Shut down the Thread pool, releasing all associated threads

  function Push(aContext: pointer; aWaitOnContention: boolean=false): boolean;
    Let a task (specified as a pointer) be processed by the Thread Pool
    - returns false if there is no idle thread available in the pool and
    Create(aQueuePendingContext=false) was used (caller should retry later); if
    aQueuePendingContext was true in Create, or IOCP is used, the supplied context will be added
    to an internal list and handled when possible
    - if aWaitOnContention is default false, returns immediately when the queue is full; set
      aWaitOnContention=true to wait up to ContentionAbortDelay ms and retry to queue the task

  property ContentionAbortCount: cardinal read fContentionAbortCount;
    How many tasks were rejected due to thread pool contention
    - if this number is high, consider setting a higher number of threads, or profile and tune the Task
      method
property ContentionAbortDelay: integer read fContentionAbortDelay write fContentionAbortDelay;

  *Milliseconds delay to reject a connection due to contention*
  - default is 5000, i.e. 5 seconds wait for some room to be available in the IOCP or
    aQueuePendingContext internal list
  - during this delay, no new connection is available (i.e. Accept is not called), so that a load
    balancer could detect the contention and switch to another instance in the pool, or a direct
    client may eventually have its connection rejected, so won't start sending data

property ContentionCount: cardinal read fContentionCount;

  *How many times the pool waited for an available slot in the queue*
  - contention won't fail immediately, but will retry until ContentionAbortDelay
  - any high number here may better increase the threads count
  - use this property and ContentionTime to compute the average contention time

property ContentionTime: Int64 read fContentionTime;

  *Total milliseconds spent waiting for an available slot in the queue*
  - contention won't fail immediately, but will retry until ContentionAbortDelay
  - any high number here requires code refactoring of the Task method

property RunningThreads: integer read fRunningThreads;

  *How many threads are currently running in this thread pool*

TSynThreadPoolTHttpServer = class(TSynThreadPool)

  *A simple Thread Pool, used for fast handling HTTP requests of a THttpServer*
  - will handle multi-connection with less overhead than creating a thread for each incoming
    request
  - will create a THttpServerResp response thread, if the incoming request is identified as HTTP/1.1
    keep alive, or HTTP body length is bigger than 1 MB

constructor Create(Server: THttpServer; NumberOfThreads: Integer=32); reintroduce;

  *Initialize a thread pool with the supplied number of threads*
  - Task() overridden method process the HTTP request set by Push()
  - up to 256 threads can be associated to a Thread Pool

THttpServerRequest = class(TObject)

  *A generic input/output structure used for HTTP server requests*
  - URL/Method/InHeaders/InContent properties are input parameters
  - OutContent/OutContentType/OutCustomHeader are output parameters

Status: integer;

  *Low-level property which may be used during requests processing*

constructor Create(aServer: THttpServerGeneric; aConnectionID: THttpServerConnectionID; aConnectionThread: TSynThread); virtual;

  *Initialize the context, associated to a HTTP server instance*

procedure AddInHeader(additionalHeader: SockString);

  *Append some lines to the InHeaders input parameter*
procedure Prepare(const aURL, aMethod, aInHeaders, aInContent, aInContentType, aRemoteIP: SockString; aUseSSL: boolean=false);

Prepare an incoming request
- will set input parameters URL/Method/InHeaders/InContent/InContentType
- will reset output parameters

property AuthenticatedUser: SockString read fAuthenticatedUser;
Contains the THttpServer-side authenticated user name, UTF-8 encoded
- e.g. when using http.sys authentication with HTTP API 2.0, the domain user name is retrieved from the supplied AccessToken
- could also be set by the THttpServerGeneric.Request() method, after proper authentication, so that it would be logged as expected

property AuthenticationStatus: THttpServerRequestAuthentication read fAuthenticationStatus;
Contains the THttpServer-side authentication status
- e.g. when using http.sys authentication with HTTP API 2.0

property ConnectionID: THttpServerConnectionID read fConnectionID;
The ID of the connection which called this execution context
- e.g. SynBidirSock's TWebSocketProcess.NotifyCallback method would use this property to specify the client connection to be notified
- is set as an Int64 to match http.sys ID type, but will be an increasing 31-bit integer sequence for (web)socket-based servers

property ConnectionThread: TSynThread read fConnectionThread;
The thread which owns the connection of this execution context
- depending on the HTTP server used, may not follow ConnectionID

property FullURL: SockUnicode read fFullURL;
Input parameter containing the caller Full URL

property HttpApiRequest: Pointer read fHttpApiRequest;
For THttpApiServer, points to a PHTTP_REQUEST structure
- not used by now for other servers

property InContent: SockString read fInContent;
Input parameter containing the caller message body
- e.g. some GET/POST/PUT JSON data can be specified here

property InContentType: SockString read fInContentType;
Input parameter defining the caller message body content type

property InHeaders: SockString read fInHeaders;
Input parameter containing the caller message headers

property Method: SockString read fMethod;
Input parameter containing the caller method (GET/POST...)

property OutContent: SockString read fOutContent write fOutContent;
Output parameter to be set to the response message body
property OutContentType: SockString read fOutContentType write fOutContentType;
    Output parameter to define the response message body content type
    - if OutContentType is HTTP_RESP_STATICFILE (i.e. 'STATICFILE', defined as STATICFILE_CONTENT_TYPE in mORMot.pas), then OutContent is the UTF-8 file name of a file which must be sent to the client via http.sys or NGINX's X-Accel-Redirect header (faster than local buffering/sending)
    - if OutContentType is HTTP_RESP_NORESPONSE (i.e. 'NORESPONSE', defined as NORESPONSE_CONTENT_TYPE in mORMot.pas), then the actual transmission protocol may not wait for any answer - used e.g. for WebSockets

property OutCustomHeaders: SockString read fOutCustomHeaders write fOutCustomHeaders;
    Output parameter to be sent back as the response message header
    - e.g. to set Content-Type/Location

property RemoteIP: SockString read fRemoteIP write fRemoteIP;
    The client remote IP, as specified to Prepare()

property RequestID: integer read fRequestID;
    A 31-bit sequential number identifying this instance on the server

property Server: THttpServerGeneric read fServer;
    The associated server instance
    - may be a THttpServer or a THttpApiServer class

property URL: SockString read fURL;
    Input parameter containing the caller URI

property UseSSL: boolean read fUseSSL;
    Is TRUE if the caller is connected via HTTPS
    - only set for THttpApiServer class yet

TServerGeneric = class(TSynThread)
    Abstract class to implement a server thread
    - do not use this class, but rather the THttpServer, THttpApiServer or TAsynchFrameServer (as defined in SynBidirSock)

constructor Create(CreateSuspended: boolean; OnStart,OnStop: TNotifyThreadEvent; const ProcessName: SockString); reintroduce; virtual;
    Initialize the server instance, in non suspended state

THttpServerGeneric = class(TServerGeneric)
    Abstract class to implement an HTTP server
    - do not use this class, but rather the THttpServer or THttpApiServer

Used for DI-2.1.1.2.4 (page 2545).

constructor Create(CreateSuspended: boolean; OnStart,OnStop: TNotifyThreadEvent; const ProcessName: SockString); reintroduce; virtual;
    31-bit internal sequence initialize the server instance, in non suspended state
**function** Callback(Ctxt: THttpServerRequest; aNonBlocking: boolean): cardinal; *virtual;*

*Server can send a request back to the client, when the connection has been upgraded e.g. to WebSockets*
- InURL/InMethod/InContent properties are input parameters (InContentType is ignored)
- OutContent/OutContentType/OutCustomHeader are output parameters
- CallingThread should be set to the client's Ctxt.CallingThread value, so that the method could know which connection is to be used - it will return STATUS_NOTFOUND (404) if the connection is unknown
- result of the function is the HTTP error code (200 if OK, e.g.)
- warning: this void implementation will raise an ECrtSocket exception - inherited classes should override it, e.g. as in TWebSocketServerRest

**function** Request(Ctxt: THttpServerRequest): cardinal; *virtual;*

*Override this function to customize your http server*
- InURL/InMethod/InContent properties are input parameters
- OutContent/OutContentType/OutCustomHeader are output parameters
- result of the function is the HTTP error code (200 if OK, e.g.),
- OutCustomHeader is available to handle Content-Type/Location
- if OutContentType is HTTP_RESP_STATICFILE (i.e. 'STATICFILE' or STATICFILE_CONTENT_TYPE defined in mORMot.pas), then OutContent is the UTF-8 file name of a file which must be sent to the client via http.sys or NGINX's X-Accel-Redirect (much faster than manual buffering/sending); the OutCustomHeader should contain the proper 'Content-type: ....'
- default implementation is to call the OnRequest event (if existing), and will return STATUS_NOTFOUND if OnRequest was not set
- warning: this process must be thread-safe (can be called by several threads simultaneously, but with a given Ctxt instance for each)

*Used for DI-2.1.1.2.4 (page 2545).*

**procedure** RegisterCompress(aFunction: THttpSocketCompress; aCompressMinSize: integer=1024); *virtual;*

*Will register a compression algorithm*
- used e.g. to compress on the fly the data, with standard gzip/deflate or custom (synlzo/synlz) protocols
- you can specify a minimal size (in bytes) before which the content won't be compressed (1024 by default, corresponding to a MTU of 1500 bytes)
- the first registered algorithm will be the preferred one for compression

**procedure** Shutdown;

*You can call this method to prepare the HTTP server for shutting down*

**property** APIVersion: string read GetAPIVersion;

*Returns the API version used by the inherited implementation*

**property** CanNotifyCallback: boolean read fCanNotifyCallback;

*TRUE if the inherited class is able to handle callbacks*
- only TWebSocketServerRest has this ability by now
property HTTPQueueLength: cardinal read GetHTTPQueueLength write SetHTTPQueueLength;

Defines request/response internal queue length
- default value if 1000, which sounds fine for most use cases
- for THttpApiServer, will return 0 if the system does not support HTTP API 2.0 (i.e. under Windows XP or Server 2003)
- for THttpServer, will shutdown any incoming accepted socket if the internal TSynThreadPool.PendingContextCount+ThreadCount exceeds this limit; each pending connection is a THttpGetSocket instance in the queue
- increase this value if you don't have any load-balancing in place, and in case of e.g. many 503 HTTP answers or if many "QueueFull" messages appear in HTTP.sys log files (normally in C:\Windows\System32\LogFiles\HTTPERR\httperr*.log) - may appear with thousands of concurrent clients accessing at once the same server - see @http://msdn.microsoft.com/en-us/library/windows/desktop/aa364501
- you can use this property with a reverse-proxy as load balancer, e.g. with nginx configured as such:

```plaintext
location / {
  proxy_pass http://balancing_upstream;
  proxy_next_upstream error timeout invalid_header http_500 http_503;
  proxy_connect_timeout 2;
  proxy_set_header Host $host;
  proxy_set_header X-Real-IP $remote_addr;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header X-Conn-ID $connection
}
```

see https://synopse.info/forum/viewtopic.php?pid=28174#p28174

property MaximumAllowedContentLength: cardinal read fMaximumAllowedContentLength write SetMaximumAllowedContentLength;

Reject any incoming request with a body size bigger than this value
- default to 0, meaning any input size is allowed
- returns STATUS_PAYLOADTOOLARGE = 413 error if "Content-Length" incoming header overflow the supplied number of bytes

property OnAfterRequest: TOnHttpServerRequest read fOnAfterRequest write SetOnAfterRequest;

Event handler called after request is processed but before response is sent back to client
- main purpose is to apply post-processor, not part of request logic
- if handler returns value > 0 it will override the OnProcess response code
- warning: this handler must be thread-safe (can be called by several threads simultaneously)

property OnAfterResponse: TOnHttpServerAfterResponse read fOnAfterResponse write SetOnAfterResponse;

Event handler called after response is sent back to client
- main purpose is to apply post-response analysis, logging, etc.
- warning: this handler must be thread-safe (can be called by several threads simultaneously)

property OnBeforeBody: TOnHttpServerBeforeBody read fOnBeforeBody write SetOnBeforeBody;

Event handler called just before the body is retrieved from the client
- should return STATUS_SUCCESS=200 to continue the process, or an HTTP error code to reject the request immediatly, and close the connection
**property** OnBeforeRequest: TOnHttpServerRequest read fOnBeforeRequest write SetOnBeforeRequest;

*Event handler called after HTTP body has been retrieved, before OnProcess*
- may be used e.g. to return a STATUS_ACCEPTED (202) status to client and continue a long-term job inside the OnProcess handler in the same thread; or to modify incoming information before passing it to main business logic, (header preprocessor, body encoding etc...)
- if the handler returns > 0 server will send a response immediately, unless return code is STATUS_ACCEPTED (202), then OnRequest will be called
- warning: this handler must be thread-safe (can be called by several threads simultaneously)

**property** OnHttpThreadStart: TNotifyThreadEvent read fOnHttpThreadStart write fOnHttpThreadStart;

*Event handler called after each working Thread is just initiated*
- called in the thread context at first place in THttpServerGeneric.Execute

**property** OnHttpThreadTerminate: TNotifyThreadEvent read fOnThreadTerminate write SetOnThreadTerminate;

*Event handler called when a working Thread is terminating*
- called in the corresponding thread context
- the TThread.OnTerminate event will be called within a Synchronize() wrapper, so it won't fit our purpose
- to be used e.g. to call CoUnInitialize from thread in which CoInitialize was made, for instance via a method defined as such:

```pascal
procedure TMyServer.OnHttpThreadTerminate(Sender: TObject);
begin
  // TSQLDBConnectionProperties.ThreadSafe
  fMyConnectionProps.EndCurrentThread;
end;
```

- is used e.g. by TSQLRest.EndCurrentThread for proper multi-threading

**property** OnRequest: TOnHttpServerRequest read fOnRequest write SetOnRequest;

*Event handler called by the default implementation of the virtual Request method*
- warning: this process must be thread-safe (can be called by several threads simultaneously)

*Used for DI-2.1.1.2.4 (page 2545).*

**property** ProcessName: SockString read fProcessName write fProcessName;

*The associated process name*

**property** RemoteConnIDHeader: SockString read fRemoteConnIDHeader write SetRemoteConnIDHeader;

*The value of a custom HTTP header containing the real client connection ID*
- by default, Ctxt.ConnectionID information will be retrieved from our socket layer - but if the server runs behind some proxy service, you should define here the HTTP header name which indicates the real remote connection, for example as 'X-Conn-ID', setting in nginx config:

```
proxy_set_header      X-Conn-ID       $connection
```

**property** RemoteIPHeader: SockString read fRemoteIPHeader write SetRemoteIPHeader;

*The value of a custom HTTP header containing the real client IP*
- by default, the RemoteIP information will be retrieved from the socket layer - but if the server runs behind some proxy service, you should define here the HTTP header name which indicates the true remote client IP value, mostly as 'X-Real-IP' or 'X-Forwarded-For'
The server name, UTF-8 encoded, e.g. 'mORMot/1.18 (Linux)'
- will be served as "Server: ..." HTTP header
- for THttpApiServer, when called from the main instance, will propagate the change to all cloned instances, and included in any HTTP API 2.0 log

```
property ServerName: SockString read fServerName write SetServerName;
```

The HTTP server API enables applications to communicate over HTTP without using Microsoft Internet Information Server (IIS). Applications can register to receive HTTP requests for particular URLs, receive HTTP requests, and send HTTP responses. The HTTP Server API includes SSL support so that applications can exchange data over secure HTTP connections without IIS. It is also designed to work with I/O completion ports.

- The HTTP Server API is supported on Windows Server 2003 operating systems and on Windows XP with Service Pack 2 (SP2). Be aware that Microsoft IIS 5 running on Windows XP with SP2 is not able to share port 80 with other HTTP applications running simultaneously.

```
THttpApiServer = class(THttpServerGeneric)
```

HTTP server using fast http.sys kernel-mode server
- The HTTP Server API enables applications to communicate over HTTP without using Microsoft Internet Information Server (IIS). Applications can register to receive HTTP requests for particular URLs, receive HTTP requests, and send HTTP responses. The HTTP Server API includes SSL support so that applications can exchange data over secure HTTP connections without IIS. It is also designed to work with I/O completion ports.
- The HTTP Server API is supported on Windows Server 2003 operating systems and on Windows XP with Service Pack 2 (SP2). Be aware that Microsoft IIS 5 running on Windows XP with SP2 is not able to share port 80 with other HTTP applications running simultaneously.

```
constructor Create(CreateSuspended: boolean; QueueName: SockUnicode=''; OnStart: TNotifyThreadEvent=nil; OnStop: TNotifyThreadEvent=nil; const ProcessName: SockString=''); reintroduce;
```

Initialize the HTTP Service
- will raise an exception if http.sys is not available e.g. before Windows XP SP2) or if the request queue creation failed
- if you override this constructor, put the AddUrl() methods within, and you can set CreateSuspended to FALSE
- if you will call AddUrl() methods later, set CreateSuspended to TRUE, then call explicitly the Resume method, after all AddUrl() calls, in order to start the server

```
destructor Destroy; override;
```

Release all associated memory and handles

```
function AddUrl(const aRoot, aPort: SockString; Https: boolean=false; const aDomainName: SockString=''; aRegisterURI: boolean=false; aContext: Int64=0): integer;
```

Register the URLs to Listen On
- e.g. AddUrl('root','888')
- aDomainName could be either a fully qualified case-insensitive domain name, an IPv4 or IPv6 literal string, or a wildcard ('+' will bound to all domain names for the specified port, '*' will accept the request when no other listening hostnames match the request for that port)
- return 0 (NO_ERROR) on success, an error code if failed: under Vista and Seven, you could have ERROR_ACCESS_DENIED if the process is not running with enough rights (by default, UAC requires administrator rights for adding an URL to http.sys registration list) - solution is to call the THttpApiServer.AddUrlAuthorize class method during program setup
- if this method is not used within an overridden constructor, default Create must have be called with CreateSuspended = TRUE and then call the Resume method after all Url have been added
- if aRegisterURI is TRUE, the URI will be registered (need administrator rights) - default is FALSE, as defined by Windows security policy
**class function AddUrlAuthorize(const aRoot, aPort: SockString; Https: boolean=false; const aDomainName: SockString='*'; OnlyDelete: boolean=false): string;**

*Will authorize a specified URL prefix*
- will allow to call AddUrl() later for any user on the computer
- if aRoot is left '', it will authorize any root for this port
- must be called with Administrator rights: this class function is to be used in a Setup program for instance, especially under Vista or Seven, to reserve the Url for the server
- add a new record to the http.sys URL reservation store
- return '' on success, an error message otherwise
- will first delete any matching rule for this URL prefix
- if OnlyDelete is true, will delete but won't add the new authorization; in this case, any error message at deletion will be returned

**function HasAPI2: boolean;**

*Can be used to check if the HTTP API 2.0 is available*

**function RemoveUrl(const aRoot, aPort: SockString; Https: boolean=false; const aDomainName: SockString='*'): integer;**

*Un-register the URLs to Listen On*
- this method expect the same parameters as specified to AddUrl()
- return 0 (NO_ERROR) on success, an error code if failed (e.g. -1 if the corresponding parameters do not match any previous AddUrl)

**procedure Clone(ChildThreadCount: integer);**

*Will clone this thread into multiple other threads*
- could speed up the process on multi-core CPU
- will work only if the OnProcess property was set (this is the case e.g. in TSQLHttpServer.Create() constructor)
- maximum value is 256 - higher should not be worth it

**procedure LogStart(const aLogFolder: TFileName; aType: THttpApiLoggingType=hltW3C; const aSoftwareName: TFileName=''; aRolloverType: THttpApiLoggingRollover=hlrDaily; aRolloverSize: cardinal=0; aLogFields: THttpApiLogFields=[hlfDate..hlfSubStatus]; aFlags: THttpApiLoggingFlags=[hlfUseUTF8Conversion]);**

*Enable HTTP API 2.0 logging*
- will raise an EHttpApiServer exception if the old HTTP API 1.x is used so you should better test the availability of the method first:
  ```pascal
  if aServer.HasAPI2 then
  LogStart(...);
  ```
- this method won't do anything on the cloned instances, but the main instance logging state will be replicated to all cloned instances
- you can select the output folder and the expected logging layout
- aSoftwareName will set the optional W3C-only software name string
- aRolloverSize will be used only when aRolloverType is hlrSize

**procedure LogStop;**

*Disable HTTP API 2.0 logging*
- this method won't do anything on the cloned instances, but the main instance logging state will be replicated to all cloned instances
**procedure** RegisterCompress(aFunction: THttpSocketCompress; aCompressMinSize: integer=1024); **override**;

Will register a compression algorithm
- overridden method which will handle any cloned instances

**procedure** SetAuthenticationSchemes(schemes: THttpApiRequestAuthentications; **const** DomainName: SockUnicode=''; **const** Realm: SockUnicode='');

*Enable HTTP API 2.0 server-side authentication*
- once enabled, the client sends an unauthenticated request: it is up to the server application to generate the initial 401 challenge with proper WWW-Authenticate headers; any further authentication steps will be perform in kernel mode, until the authentication handshake is finalized; later on, the application can check the AuthenticationStatus property of THttpServerRequest and its associated AuthenticatedUser value see https://msdn.microsoft.com/en-us/library/windows/desktop/aa364452
- will raise an EHttpApiServer exception if the old HTTP API 1.x is used so you should better test the availability of the method first:
  ```delphi```
  if aServer.HasAPI2 then
    SetAuthenticationSchemes(....);
  ```delphi```
- this method will work on the current group, for all instances
- see HTTPAPI_AUTH_ENABLE_ALL constant to set all available schemes
- optional Realm parameters can be used when haBasic scheme is defined
- optional DomainName and Realm parameters can be used for haDigest

**procedure** SetTimeOutLimits(aEntityBody, aDrainEntityBody, aRequestQueue, aIdleConnection, aHeaderWait, aMinSendRate: cardinal);

*Enable HTTP API 2.0 advanced timeout settings*
- all those settings are set for the current URL group
- will raise an EHttpApiServer exception if the old HTTP API 1.x is used so you should better test the availability of the method first:
  ```delphi```
  if aServer.HasAPI2 then
    SetTimeOutLimits(....);
  ```delphi```
- `aEntityBody` is the time, in seconds, allowed for the request entity body to arrive - default value is 2 minutes
- `aDrainEntityBody` is the time, in seconds, allowed for the HTTP Server API to drain the entity body on a Keep-Alive connection - default value is 2 minutes
- `aRequestQueue` is the time, in seconds, allowed for the request to remain in the request queue before the application picks it up - default value is 2 minutes
- `aIdleConnection` is the time, in seconds, allowed for an idle connection; is similar to THttpServer.ServerKeepAliveTimeOut - default value is 2 minutes
- `aHeaderWait` is the time, in seconds, allowed for the HTTP Server API to parse the request header - default value is 2 minutes
- `aMinSendRate` is the minimum send rate, in bytes-per-second, for the response - default value is 150 bytes-per-second
- any value set to 0 will set the HTTP Server API default value

**property** AuthenticationSchemes: THttpApiRequestAuthentications **read** fAuthenticationSchemes;

*Read-only access to HTTP API 2.0 server-side enabled authentication schemes*

**property** Cloned: boolean **read** GetCloned;

*TRUE if this instance is in fact a cloned instance for the thread pool*
property Clones: THttpApiServers read fClones;
  Access to the internal THttpApiServer list cloned by this main instance
  - as created by Clone() method

property Logging: boolean read GetLogging;
  Read-only access to check if the HTTP API 2.0 logging is enabled
  - use LogStart/LogStop methods to change this property value

property LoggingServiceName: SockString read fLoggingServiceName write SetLoggingServiceName;
  The current HTTP API 2.0 logging Service name
  - should be UTF-8 encoded, if LogStart(aFlags=[hlfUseUTF8Conversion])
  - this value is dedicated to one instance, so the main instance won't propagate the change to all
    cloned instances

property MaxBandwidth: Cardinal read GetMaxBandwidth write SetMaxBandwidth;
  The maximum allowed bandwidth rate in bytes per second (via HTTP API 2.0)
  - Setting this value to 0 allows an unlimited bandwidth
  - by default Windows not limit bandwidth (actually limited to 4 Gbit/sec).
  - will return 0 if the system does not support HTTP API 2.0 (i.e. under Windows XP or Server
    2003)

property MaxConnections: Cardinal read GetMaxConnections write SetMaxConnections;
  The maximum number of HTTP connections allowed (via HTTP API 2.0)
  - Setting this value to 0 allows an unlimited number of connections
  - by default Windows does not limit number of allowed connections
  - will return 0 if the system does not support HTTP API 2.0 (i.e. under Windows XP or Server
    2003)

property ReceiveBufferSize: cardinal read fReceiveBufferSize write SetReceiveBufferSize;
  How many bytes are retrieved in a single call to ReceiveRequestEntityBody
  - set by default to 1048576, i.e. 1 MB - practical limit is around 20 MB
  - you may customize this value if you encounter HTTP error STATUS_NOTACCEPTABLE (406)
  - from client, corresponding to an ERROR_NO_SYSTEM_RESOURCES (1450) exception on server
    side, when uploading huge data content

property RegisteredUrl: SockUnicode read GetRegisteredUrl;
  Return the list of registered URL on this server instance

property SessionID: HTTP_SERVER_SESSION_ID read fServerSessionID;
  Read-only access to the low-level HTTP API 2.0 Session ID

property UrlGroupID: HTTP_URL_GROUP_ID read fUrlGroupID;
  Read-only access to the low-level HTTP API 2.0 URI Group ID

THttpApiWebSocketConnection = object(TObject)
  Structure representing a single WebSocket connection

procedure Close(aStatus: WEB_SOCKET_CLOSE_STATUS; aBuffer: Pointer; aBufferSize: ULONG);
  Close connection
procedure Send(aBufferType: WEB_SOCKET_BUFFER_TYPE; aBuffer: Pointer; aBufferSize: ULONG);

Send data to client

property Index: integer read fIndex;
Index of connection in protocol's connection list

property PrivateData: pointer read fPrivateData write fPrivateData;
Custom user data

Protocol of connection

property State: TWebSocketState read fState;
Access to the current state of this connection

THttpApiWebSocketServerProtocol = class(TObject)
Protocol Handler of websocket endpoints events
- maintains a list of all WebSockets clients for a given protocol

initialize the WebSockets process
- if aManualFragmentManagement is true, onMessage will appear only for whole received messages, otherwise OnFragment handler must be passed (for video broadcast, for example)

destructor Destroy; override;
Finalize the process

function Broadcast(aBufferType: ULONG; aBuffer: Pointer; aBufferSize: ULONG): boolean;
Send message to all connections of this protocol

function Close(index: Integer; aStatus: WEB_SOCKET_CLOSE_STATUS; aBuffer: Pointer; aBufferSize: ULONG): boolean;
Close WebSocket connection identified by its index

function Send(index: Integer; aBufferType: ULONG; aBuffer: Pointer; aBufferSize: ULONG): boolean;
Send message to the WebSocket connection identified by its index

property Index: integer read fIndex;
Identify the endpoint instance

property ManualFragmentManagement: Boolean read fManualFragmentManagement;
OnFragment event will be called for each fragment

property Name: SockString read fName;
Text identifier
property OnAccept: THttpApiWebSocketServerOnAcceptEvent read fOnAccept;
   Event triggered when a WebSockets client is initiated

property OnConnect: THttpApiWebSocketServerOnConnectEvent read fOnConnect;
   Event triggered when a WebSockets client is connected

property OnDisconnect: THttpApiWebSocketServerOnDisconnectEvent read fOnDisconnect;
   Event triggered when a WebSockets client is gracefully disconnected

property OnFragment: THttpApiWebSocketServerOnMessageEvent read fOnFragment;
   Event triggered when a non complete frame is received
   - required if ManualFragmentManagement is true

   Event triggered when a WebSockets message is received

THttpApiWebSocketServer = class(THttpApiServer)
   HTTP & WebSocket server using fast http.sys kernel-mode server
   - can be used like simple THttpApiServer
   - when AddUrlWebSocket is called WebSocket support are added in this case WebSocket will
     receiving the frames in asynchronous

constructor Create(CreateSuspended: Boolean; aSocketThreadsCount: integer=1;
   aPingTimeout: integer=0; QueueName: SockUnicode=''; aOnWSThreadStart:
   TNotifyThreadEvent=nil; aOnWSThreadTerminate: TNotifyThreadEvent=nil);
   reintroduce;
   Initialize the HTTPAPI based Server with WebSocket support
   - will raise an exception if http.sys or websocket.dll is not available (e.g. before Windows 8) or if
   the request queue creation failed
   - for aPingTimeout explanation see PingTimeout property documentation

function AddUrlWebSocket(const aRoot, aPort: SockString; Https: boolean=false;
   const aDomainName: SockString='*'; aRegisterURI: boolean=false): integer;
   Register the URLs to Listen on using WebSocket
   - aProtocols is an array of a recond with callbacks, server call during WebSocket activity

procedure RegisterProtocol(const aName: SockString; aManualFragmentManagement:
   Boolean; aOnAccept: THttpApiWebSocketServerOnAcceptEvent; aOnMessage:
   THttpApiWebSocketServerOnMessageEvent; aOnConnect:
   THttpApiWebSocketServerOnConnectEvent; aOnDisconnect:
   THttpApiWebSocketServerOnDisconnectEvent; aOnFragment:
   THttpApiWebSocketServerOnMessageEvent=nil);
   Prepare the process for a given THttpApiWebSocketServerProtocol

procedure SendServiceMessage;
   Can be called from any thread
   - will send a "service" message to a WebSocketServer to wake up a WebSocket thread
   - When a WebSocket thread receives such a message it will call onServiceMessage in the thread
     context
property OnServiceMessage: TThreadMethod read fOnServiceMessage write fOnServiceMessage;
    Event called when a service message is raised

property OnWSThreadStart: TNotifyThreadEvent read FOnWSThreadStart write SetOnWSThreadStart;
    Event called when the processing thread starts

property OnWSThreadTerminate: TNotifyThreadEvent read FOnWSThreadTerminate write SetOnWSThreadTerminate;
    Event called when the processing thread terminates

property PingTimeout: integer read fPingTimeout;
    Ping timeout in seconds. 0 mean no ping.
    - if connection not receive messages longer than this timeout TSynWebSocketGuard will send
      ping frame
    - if connection not receive any messages longer than double of this timeout it will be closed

    Access to the associated endpoints

property ProtocolsCount: Integer read getProtocolsCount;
    Access to the associated endpoints count

TSynThreadPoolHttpApiWebSocketServer = class(TSynThreadPool)
    A Thread Pool, used for fast handling WebSocket requests

constructor Create(Server: THttpApiWebSocketServer; NumberOfThreads: Integer=1);
reintroduce;
    Initialize the thread pool

TSynWebSocketGuard = class(TThread)
    Thread for closing WebSocket connections which not response more than PingTimeout interval

constructor Create(Server: THttpApiWebSocketServer); reintroduce;
    Initialize the thread
Main HTTP server Thread using the standard Sockets API (e.g. WinSock)
- bind to a port and listen to incoming requests
- assign this requests to THttpServerResp threads from a ThreadPool
- it implements a HTTP/1.1 compatible server, according to RFC 2068 specifications
- if the client is also HTTP/1.1 compatible, KeepAlive connection is handled: multiple requests will use the existing connection and thread; this is faster and uses less resources, especially under Windows
- a Thread Pool is used internally to speed up HTTP/1.0 connections - a typical use, under Linux, is to run this class behind a NGINX frontend, configured as https reverse proxy, leaving default "proxy_http_version 1.0" and "proxy_request_buffering on" options for best performance, and setting KeepAliveTimeOut=0 in the THttpServer.Create constructor
- under windows, will trigger the firewall UAC popup at first run
- don't forget to use Free method when you are finished

Used for DI-2.1.1.2.4 (page 2545).

create a Server Thread, ready to be bound and listening on a port
- this constructor will raise a EHttpServer exception if binding failed
- expects the port to be specified as string, e.g. '1234'; you can optionally specify a server address to bind to, e.g. '1.2.3.4:1234'
- can listed on UDS in case port is specified with 'unix:' prefix, e.g. 'unix:/run/myapp.sock'
- on Linux in case aPort is empty string will check if external fd is passed by systemd and use it (so called systemd socked activation)
- you can specify a number of threads to be initialized to handle incoming connections. Default is 32, which may be sufficient for most cases, maximum is 256. If you set 0, the thread pool will be disabled and one thread will be created for any incoming connection
- you can also tune (or disable with 0) HTTP/1.1 keep alive delay and how incoming request Headers[] are pushed to the processing method
- this constructor won't actually do the port binding, which occurs in the background thread: caller should therefore call WaitForStarted after THttpServer.Create()

Used for DI-2.1.1.2.4 (page 2545).

Release all memory and handlers
procedure NginxSendFileFrom(const FileNameLeftTrim: TFileName);

Enable NGINX X-Accel internal redirection for HTTP_RESP_STATICFILE
- will define internally a matching OnSendFile event handler
- generating "X-Accel-Redirect: " header, trimming any supplied left case-sensitive file name prefix, e.g. with NginxSendFileFrom('/var/www'):

```ruby
# Will serve /var/www/protected_files/myfile.tar.gz
# When passed URI /protected_files/myfile.tar.gz
location /protected_files {
  internal;
  root /var/www;
}
```
- call this method several times to register several folders

procedure WaitStarted(Seconds: integer = 30); virtual;

Ensure the HTTP server thread is actually bound to the specified port
- TCrtSocket.Bind() occurs in the background in the Execute method: you should call and check this method result just after THttpServer.Create
- initial THttpServer design was to call Bind() within Create, which works fine on Delphi + Windows, but fails with a EThreadError on FPC/Linux
- raise a ECrtSocket if binding failed within the specified period (if port is free, it would be almost immediate)
- calling this method is optional, but if the background thread didn’t actually bind the port, the server will be stopped and unresponsive with no explicit error message, until it is terminated

property HeaderRetrieveAbortDelay: integer read fHeaderRetrieveAbortDelay write fHeaderRetrieveAbortDelay;

Milliseconds delay to reject a connection due to too long header retrieval
- default is 0, i.e. not checked (typically not needed behind a reverse proxy)

property HeadersNotFiltered: boolean read fHeadersNotFiltered;

By default, only relevant headers are added to internal headers list
- for instance, Content-Length, Content-Type and Content-Encoding are stored as fields in this THttpSocket, but not included in its Headers[]
- set this property to true to include all incoming headers

property OnSendFile: TOnHttpServerSendFile read fOnSendFile write fOnSendFile;

Custom event handler used to send a local file for HTTP_RESP_STATICFILE
- see also NginxSendFileFrom() method

property ServerConnectionActive: integer read fServerConnectionActive write fServerConnectionActive;

Will contain the current number of connections to the server

property ServerConnectionCount: integer read fServerConnectionCount write fServerConnectionCount;

Will contain the total number of connections to the server
- it's the global count since the server started
   Time, in milliseconds, for the HTTP/1.1 connections to be kept alive
   - default is 30000 ms, i.e. 30 seconds
   - setting 0 here (or in KeepAliveTimeOut constructor parameter) will disable keep-alive, and
     fallback to HTTP/1.0 for all incoming requests (may be a good idea e.g. behind a NGINX reverse
     proxy)
   - see THttpApiServer.SetTimeOutLimits(aIdleConnection) parameter

property Sock: TCrtSocket read fSock;
   Access to the main server low-level Socket
   - it's a raw TCrtSocket, which only need a socket to be bound, listening and accept incoming
     request
   - THttpServerSocket are created on the fly for every request, then a THttpServerResp thread is
     created for handling this THttpServerSocket

property SockPort: SockString read fSockPort;
   The bound TCP port, as specified to Create() constructor
   - TCrtSocket.Bind() occurs in the Execute method

property StatBodyProcessed: integer index grBodyReceived read GetStat;
   How many HTTP bodies have been processed

property StatHeaderErrors: integer index grError read GetStat;
   How many invalid HTTP headers have been rejected

property StatHeaderException: integer index grException read GetStat;
   How many invalid HTTP headers raised an exception

property StatHeaderProcessed: integer index grHeaderReceived read GetStat;
   How many HTTP headers have been processed

property StatHeaderTimeout: integer index grTimeout read GetStat;
   How many HTTP requests were rejected after HeaderRetrieveAbortDelay timeout

property StatOversizedPayloads: integer index grOversizedPayload read GetStat;
   How many HTTP requests pushed more than MaximumAllowedContentLength bytes

property StatOwnedConnections: integer index grOwned read GetStat;
   How many HTTP connections were passed to an asynchronous handler
   - e.g. for background WebSockets processing after proper upgrade

property StatRejected: integer index grRejected read GetStat;
   How many HTTP requests were rejected by the OnBeforeBody event handler

property TCPPrefix: SockString read fTCPPrefix write fTCPPrefix;
   TCP/IP prefix to mask HTTP protocol
   - if not set, will create full HTTP/1.0 or HTTP/1.1 compliant content
   - in order to make the TCP/IP stream not HTTP compliant, you can specify a prefix which will be
     put before the first header line: in this case, the TCP/IP stream won't be recognized as HTTP, and
     will be ignored by most AntiVirus programs, and increase security - but you won't be able to use
     an Internet Browser nor AJAX application for remote access any more
property ThreadPool: TSynThreadPoolTHttpServer read fThreadPool;
    // The associated thread pool
    // - may be nil if ServerThreadPoolCount was 0 on constructor

TURI = object(TObject)
    // Structure used to parse an URI into its components
    // - ready to be supplied e.g. to a THttpRequest sub-class
    // - used e.g. by class function THttpRequest.Get()
    // - will decode standard HTTP/HTTPS urls or Unix sockets URI like
    //   'http://unix:/path/to/socket.sock:/url/path'

    Address: SockString;
        // The resource address, including optional parameters
        // - e.g. '/category/name/10?param=1'

   Https: boolean;
        // If the server is accessible via https:// and not plain http://

   Layer: TCrtSocketLayer;
        // Either cslTcp for HTTP/HTTPS or cslUnix for Unix socket URI

    Port: SockString;
        // The server port
        // - e.g. '80'

    Scheme: SockString;
        // If the server is accessible via something else than http:// or https://
        // - e.g. 'ws' or 'wss' for ws:// or wss://

    Server: SockString;
        // The server name
        // - e.g. 'www.somewebsite.com' or 'path/to/socket.sock' Unix socket URI

function From(aURI: SockString; const DefaultPort: SockString=''): boolean;
    // Fill the members from a supplied URI
    // - recognize e.g. 'http://Server:Port/Address', 'https://Server/Address', 'Server/Address' (as http),
    //   or 'http://unix:/Server:/Address'
    // - returns TRUE if at least the Server has been extracted, FALSE on error

function PortInt: integer;
    // The server port, as integer value

function Root: SockString;
    // Compute the root resource Address, without any URI-encoded parameter
    // - e.g. '/category/name/10'

function URI: SockString;
    // Compute the whole normalized URI
    // - e.g. 'https://Server:Port/Address' or 'http://unix:/Server:/Address'

procedure Clear;
    // Reset all stored information
THttpRequestExtendedOptions = record
  A record to set some extended options for HTTP clients
  - allow easy propagation e.g. from a TSQLHttpClient wrapper class to the actual SynCrtSock's
    THttpRequest implementation class

Auth: record
  Allow HTTP authentication to take place at connection
  - Auth.Scheme and UserName/Password properties are handled by the TWinHttp class only by now

IgnoreSSLCertificateErrors: boolean;
  Let HTTPS be less paranoid about SSL certificates
  - IgnoreSSLCertificateErrors is handled by TWinHttp and TCurlHTTP

UserAgent: SockString;
  Allow to customize the User-Agent header

THttpRequest = class(TObject)
  To have existing RTTI for published properties abstract class to handle HTTP/1.1 request
  - never instantiate this class, but inherited TWinHTTP, TWinINet or TCurlHTTP

constructor Create(const aURI: SockString; const aProxyName: SockString=''; const aProxyByPass: SockString=''; ConnectionTimeOut: DWORD=0; SendTimeout: DWORD=0; ReceiveTimeout: DWORD=0; aIgnoreSSLCertificateErrors: boolean=false); overload;
  - Connect to the supplied URI
    - is just a wrapper around TURI and the overloaded Create() constructor

constructor Create(const aServer, aPort: SockString; aHttps: boolean; const aProxyName: SockString=''; const aProxyByPass: SockString=''; ConnectionTimeOut: DWORD=0; SendTimeout: DWORD=0; ReceiveTimeout: DWORD=0; aLayer: TCrtSocketLayer=cslTCP); overload; virtual;
  - Connect to http://aServer:aPort or https://aServer:aPort
    - optional aProxyName may contain the name of the proxy server to use, and aProxyByPass an
      optional semicolon delimited list of host names or IP addresses, or both, that should not be
      routed through the proxy: aProxyName/aProxyByPass will be recognized by TWinHTTP and
      TWinINet, and aProxyName will set the CURLOPT_PROXY option to TCurlHttp (see
      https://curl.haxx.se/libcurl/c/CURLOPT_PROXY.html as reference)
    - you can customize the default client timeouts by setting appropriate SendTimeout and
      ReceiveTimeout parameters (in ms) - note that after creation of this instance, the connection is
      tied to the initial parameters, so we won't publish any properties to change those initial values
      once created - if you left the 0 default parameters, it would use global
      HTTP_DEFAULT_CONNECTTIMEOUT, HTTP_DEFAULT_SENDTIMEOUT and
      HTTP_DEFAULT_RECEIVETIMEOUT variable values
    - *TimeOut parameters are currently ignored by TCurlHttp
class function Delete(const aURI: SockString; const aHeader: SockString='';
                        aIgnoreSSLCertificateErrors: boolean=true; outHeaders: PSockString=nil; outStatus: PInteger=nil): SockString;

Wrapper method to delete a resource via an HTTP DELETE
- will parse the supplied URI to check for the http protocol (HTTP/HTTPS), server name and port, and resource name
- aIgnoreSSLCertificateErrors will ignore the error when using untrusted certificates
- it will internally create a THttpRequest inherited instance: do not use THttpRequest.Delete() but either TWinHTTP.Delete(), TWinINet.Delete() or TCurlHTTP.Delete() methods

class function Get(const aURI: SockString; const aHeader: SockString='';
                       aIgnoreSSLCertificateErrors: boolean=true; outHeaders: PSockString=nil; outStatus: PInteger=nil): SockString;

Wrapper method to retrieve a resource via an HTTP GET
- will parse the supplied URI to check for the http protocol (HTTP/HTTPS), server name and port, and resource name
- aIgnoreSSLCertificateErrors will ignore the error when using untrusted certificates
- it will internally create a THttpRequest inherited instance: do not use THttpRequest.Get() but either TWinHTTP.Get(), TWinINet.Get() or TCurlHTTP.Get() methods

class function IsAvailable: boolean; virtual; abstract;

Returns TRUE if the class is actually supported on this system

class function Post(const aURI, aData: SockString; const aHeader: SockString='';
                       aIgnoreSSLCertificateErrors: boolean=true; outHeaders: PSockString=nil; outStatus: PInteger=nil): SockString;

Wrapper method to create a resource via an HTTP POST
- will parse the supplied URI to check for the http protocol (HTTP/HTTPS), server name and port, and resource name
- aIgnoreSSLCertificateErrors will ignore the error when using untrusted certificates
- the supplied aData content is POSTed to the server, with an optional aHeader content
- it will internally create a THttpRequest inherited instance: do not use THttpRequest.Post() but either TWinHTTP.Post(), TWinINet.Post() or TCurlHTTP.Post() methods

class function Put(const aURI, aData: SockString; const aHeader: SockString='';
                       aIgnoreSSLCertificateErrors: boolean=true; outHeaders: PSockString=nil; outStatus: PInteger=nil): SockString;

Wrapper method to update a resource via an HTTP PUT
- will parse the supplied URI to check for the http protocol (HTTP/HTTPS), server name and port, and resource name
- aIgnoreSSLCertificateErrors will ignore the error when using untrusted certificates
- the supplied aData content is PUT to the server, with an optional aHeader content
- it will internally create a THttpRequest inherited instance: do not use THttpRequest.Put() but either TWinHTTP.Put(), TWinINet.Put() or TCurlHTTP.Put() methods
function RegisterCompress(aFunction: THttpSocketCompress; aCompressMinSize: integer=1024): boolean;
  Will register a compression algorithm
  - used e.g. to compress on the fly the data, with standard gzip/deflate or custom (synlzo/synlz) protocols
  - returns true on success, false if this function or this ACCEPT-ENCODING: header was already registered
  - you can specify a minimal size (in bytes) before which the content won't be compressed (1024 by default, corresponding to a MTU of 1500 bytes)
  - the first registered algorithm will be the preferred one for compression

function Request(const url, method: SockString; KeepAlive: cardinal; const InHeader, InData, InDataType: SockString; out OutHeader, OutData: SockString): integer;
  Low-level HTTP/1.1 request
  - after an Create(server,port), return 200,202,204 if OK, http status error otherwise
  - KeepAlive is in milliseconds, 0 for "Connection: Close" HTTP/1.0 requests

  Optional Password for Authentication

property AuthScheme: THttpRequestAuthentication read fExtendedOptions.Auth.Scheme write fExtendedOptions.Auth.Scheme;
  Optional Authentication Scheme

property AuthUserName: SockUnicode read fExtendedOptions.Auth.UserName write fExtendedOptions.Auth.UserName;
  Optional User Name for Authentication

property ExtendedOptions: THttpRequestExtendedOptions read fExtendedOptions write fExtendedOptions;
  Internal structure used to store extended options
  - will be replicated by IgnoreSSLCertificateErrors and Auth* properties

property Https: boolean read fHttps;
  If the remote server uses HTTPS, as specified to the class constructor

property IgnoreSSLCertificateErrors: boolean read fExtendedOptions.IgnoreSSLCertificateErrors write fExtendedOptions.IgnoreSSLCertificateErrors;
  Allows to ignore untrusted SSL certificates
  - similar to adding a security exception for a domain in the browser

property Port: cardinal read fPort;
  The remote server port number, as specified to the class constructor

property ProxyByPass: SockString read fProxyByPass;
  The remote server optional proxy by-pass list, as specified to the class constructor

property ProxyName: SockString read fProxyName;
  The remote server optional proxy, as specified to the class constructor

property Server: SockString read fServer;
  The remote server host name, as stated specified to the class constructor
property Tag: PtrInt read fTag write fTag;
    Some internal field, which may be used by end-user code

property UserAgent: SockString read fExtendedOptions.UserAgent write
fExtendedOptions.UserAgent;
    Custom HTTP "User Agent:" header value

TWinHttpAPI = class(THttpRequest)
    A class to handle HTTP/1.1 request using either WinINet or WinHTTP API
    - both APIs have a common logic, which is encapsulated by this parent class
    - this abstract class defined some abstract methods which will be implemented by TWinINet or
TWinHttp with the proper API calls
    class function IsAvailable: boolean; override;
        Returns TRUE if the class is actually supported on this system

property NoAllAccept: boolean read fNoAllAccept write fNoAllAccept;
    Do not add "Accept: */*" HTTP header by default

property OnDownload: TWinHttpDownload read fOnDownload write fOnDownload;
    Download would call this method instead of filling Data: SockString value
    - may be used e.g. when downloading huge content, and saving directly the incoming data on
disk or database
    - if this property is set, raw TCP/IP incoming data would be supplied: compression and encoding
won't be handled by the class

property OnDownloadChunkSize: cardinal read fOnDownloadChunkSize write
fOnDownloadChunkSize;
    How many bytes should be retrieved for each OnDownload event chunk
    - if default 0 value is left, would use 65536, i.e. 64KB

property OnProgress: TWinHttpProgress read fOnProgress write fOnProgress;
    Download would call this method to notify progress of incoming data

property OnUpload: TWinHttpUpload read fOnUpload write fOnUpload;
    Upload would call this method to notify progress of outgoing data
    - and optionally abort sending the data by returning FALSE

TWinINet = class(TWinHttpAPI)
    A class to handle HTTP/1.1 request using the WinINet API
    - The Microsoft Windows Internet (WinINet) application programming interface (API) enables
applications to access standard Internet protocols, such as FTP and HTTP/HTTPS, similar to what IE
offers
    - by design, the WinINet API should not be used from a service, since this API may require
end-user GUI interaction
    - note: WinINet is MUCH slower than THttpClientSocket or TWinHttp: do not use this, only if you
find some configuration benefit on some old networks (e.g. to display the dialup popup window
for a GUI client application)
**destroy** Destroy; **override**

Release the connection

EWinINet = **class**(ECrtSocket)

WinINet exception type

**constructor** Create;

Create a WinINet exception, with the error message as text

TWinHTTP = **class**(TWinHttpAPI)

A class to handle HTTP/1.1 request using the WinHTTP API
- has a common behavior as THttpClientSocket() but seems to be faster over a network and is able to retrieve the current proxy settings (if available) and handle secure https connection - so it seems to be the class to use in your client programs
- WinHTTP does not share any proxy settings with Internet Explorer. The WinHTTP proxy configuration is set by either
  proxcfg.exe
  on Windows XP and Windows Server 2003 or earlier, either
  netsh.exe
  on Windows Vista and Windows Server 2008 or later; for instance, you can run either:
  proxcfg -u
  netsh winhttp import proxy source=ie
to use the current user’s proxy settings for Internet Explorer (under 64-bit Vista/Seven, to configure applications using the 32 bit WinHttp settings, call netsh or proxcfg bits from %SystemRoot%\SysWOW64 folder explicitely)
- Microsoft Windows HTTP Services (WinHTTP) is targeted at middle-tier and back-end server applications that require access to an HTTP client stack

**destroy** Destroy; **override**

Release the connection

EWinHTTP = **class**(Exception)

WinHTTP exception type

TWinHTTPUpgradeable = **class**(TWinHTTP)

A class to establish a client connection to a WebSocket server using Windows API
- used by TWinWebSocketClient class

**constructor** Create(const aServer, aPort: SockString; aHttps: boolean; const aProxyName: SockString=''; const aProxyByPass: SockString=''; ConnectionTimeOut: DWORD=0; SendTimeout: DWORD=0; ReceiveTimeout: DWORD=0; aLayer: TCrtSocketLayer=csILTCP); **override**

Initialize the instance

TWinHTTPWebSocketClient = **class**(TObject)

WebSocket client implementation
constructor Create(const aServer, aPort: SockString; aHttps: boolean; const url: SockString; const aSubProtocol: SockString = ''; const aProxyName: SockString = ''; const aProxyByPass: SockString = ''; ConnectionTimeOut: DWORD = 0; SendTimeout: DWORD = 0; ReceiveTimeout: DWORD = 0);

Initialize the instance
- all parameters do match TWinHTTP.Create except url: address of WebSocketServer for sending upgrade request

function CloseConnection(const aCloseReason: SockString): DWORD;

Close current connection

function Receive(aBuffer: pointer; aBufferLength: DWORD; out aBytesRead: DWORD; out aBufferType: WINHTTP_WEB_SOCKET_BUFFER_TYPE): DWORD;

Receive buffer

function Send(aBufferType: WINHTTP_WEB_SOCKET_BUFFER_TYPE; aBuffer: pointer; aBufferLength: DWORD): DWORD;

Send buffer

TSimpleHttpClient = class(TObject)
Simple wrapper around THttpClientSocket/THttpRequest instances
- this class will reuse the previous connection if possible, and select the best connection class available on this platform for a given URI

constructor Create(aOnlyUseClientSocket: boolean=false); reintroduce;

Initialize the instance

destructor Destroy; override;

Finalize the connection

function RawRequest(const Uri: TURI; const Method, Header, Data, DataType: SockString; KeepAlive: cardinal): integer; overload;

Low-level entry point of this instance


Simple-to-use entry point of this instance
- use Body and Headers properties to retrieve the HTTP body and headers

property Body: SockString read fBody;

Returns the HTTP body as returned by a previous call to Request()

property Headers: SockString read fHeaders;

Returns the HTTP headers as returned by a previous call to Request()

property IgnoreSSLCertificateErrors: boolean read fIgnoreSSLCertificateErrors write fIgnoreSSLCertificateErrors;

Allows to customize HTTPS connection and allow weak certificates

property Proxy: SockString read fProxy write fProxy;

Allows to customize the connection using a proxy
**property** UserAgent: SockString read fUserAgent write fUserAgent;

Allows to customize the user-agent header

**TSMTMPConnection = object(TObject)**

*May be used to store a connection to a SMTP server*

- see SendEmail() overloaded function

Host: SockString;

*The SMTP server IP or host name*

Pass: SockString;

*The SMTP user password (if any)*

Port: SockString;

*The SMTP server port (25 by default)*

User: SockString;

*The SMTP user login (if any)*

**function FromText(const aText: SockString): boolean;**

*Fill the SMTP server information from a single text field*

- expects 'user:password@smtpserver:port' format
- if aText equals SMTP_DEFAULT ('user:password@smtpserver:port'), does nothing

**TMacAddress = record**

*Interface name/address pairs as returned by GetMacAddresses*

address: SockString;

*Contains e.g. '12:50:b6:1e:c6:aa' from /sys/class/net/eth0/address*

name: SockString;

*Contains e.g. 'eth0' on Linux*

**TPollSocketResult = record**

*Modifications notified by TPollSocketAbstract.WaitForModified*

events: TPollSocketEvents;

*The events which are notified*

tag: TPollSocketTag;

*Opaque value as defined by TPollSocketAbstract.Subscribe*

**TPollSocketAbstract = class(TObject)**

*Abstract parent class for efficient socket polling*

- works like Linux epoll API in level-triggered (LT) mode
- implements libevent-like cross-platform features
- use PollSockClass global function to retrieve the best class depending on the running Operating System*
constructor Create; virtual;
  Initialize the polling

class function New: TPollSocketAbstract;
  Class function factory, returning a socket polling instance matching at best the current operating system
  - returns a TPollSocketSelect/TPollSocketPoll instance under Windows, a TPollSocketEpoll instance under Linux, or a TPollSocketPoll on BSD
  - just a wrapper around PollSockClass.Create

function Subscribe(socket: TSocket; events: TPollSocketEvents; tag: TPollSocketTag): boolean; virtual; abstract;
  Track status modifications on one specified TSocket
  - you can specify which events are monitored - pseError and pseClosed will always be notified
  - tag parameter will be returned as TPollSocketResult - you may set here the socket file descriptor value, or a transtyped class instance
  - similar to epoll's EPOLL_CTL_ADD control interface

function Unsubscribe(socket: TSocket): boolean; virtual; abstract;
  Stop status modifications tracking on one specified TSocket
  - the socket should have been monitored by a previous call to Subscribe()
  - on success, returns true and fill tag with the associated opaque value
  - similar to epoll's EPOLL_CTL_DEL control interface

function WaitForModified(out results: TPollSocketResults; timeoutMS: integer): integer; virtual; abstract;
  Waits for status modifications of all tracked TSocket
  - will wait up to timeoutMS milliseconds, 0 meaning immediate return and -1 for infinite blocking
  - returns -1 on error (e.g. no TSocket currently registered), or the number of modifications stored in results[] (may be 0 if none)

property Count: integer read fCount;
  How many TSocket instances are currently tracked

property MaxSockets: integer read fMaxSockets;
  How many TSocket instances could be tracked, at most
  - depends on the API used

TPollSocketSelect = class(TPollSocketAbstract)
  Socket polling via Windows' Select() API
  - under Windows, Select() handles up to 64 TSocket, and is available in Windows XP, whereas WSAPoll() is available only since Vista
  - under Linux, select() is very limited, so poll/epoll APIs are to be used
  - in practice, TPollSocketSelect is slightly FASTER than TPollSocketPoll when tracking a lot of connections (at least under Windows): WSAPoll() seems to be just an emulation API - very disappointing :(

constructor Create; override;
  Initialize the polling via creating an epoll file descriptor
function Subscribe(socket: TSocket; events: TPollSocketEvents; tag: TPollSocketTag): boolean; override;
    Track status modifications on one specified TSocket
    - you can specify which events are monitored - pseError and pseClosed will always be notified

function Unsubscribe(socket: TSocket): boolean; override;
    Stop status modifications tracking on one specified TSocket
    - the socket should have been monitored by a previous call to Subscribe()

function WaitForModified(out results: TPollSocketResults; timeoutMS: integer): integer; override;
    Waits for status modifications of all tracked TSocket
    - will wait up to timeoutMS milliseconds, 0 meaning immediate return and -1 for infinite blocking
    - returns -1 on error (e.g. no TSocket currently registered), or the number of modifications stored in results[] (may be 0 if none)

TPollSocketPoll = class(TPollSocketAbstract)
    Socket polling via poll/WSAPoll API
    - direct call of the Linux/POSIX poll() API, or Windows WSAPoll() API

constructor Create; override;
    Initialize the polling using poll/WSAPoll API

function Subscribe(socket: TSocket; events: TPollSocketEvents; tag: TPollSocketTag): boolean; override;
    Track status modifications on one specified TSocket
    - you can specify which events are monitored - pseError and pseClosed will always be notified

function Unsubscribe(socket: TSocket): boolean; override;
    Stop status modifications tracking on one specified TSocket
    - the socket should have been monitored by a previous call to Subscribe()

function WaitForModified(out results: TPollSocketResults; timeoutMS: integer): integer; override;
    Waits for status modifications of all tracked TSocket
    - will wait up to timeoutMS milliseconds, 0 meaning immediate return and -1 for infinite blocking
    - returns -1 on error (e.g. no TSocket currently registered), or the number of modifications stored in results[] (may be 0 if none)

TPollSockets = class(TObject)
    Implements efficient polling of multiple sockets
    - will maintain a pool of TPollSocketAbstract instances, to monitor incoming data or outgoing availability for a set of active connections
    - call Subscribe/Unsubscribe to setup the monitored sockets
    - call GetOne from any consuming threads to process new events
constructor Create(aPollClass: TPollSocketClass=nil);

Initialize the sockets polling
- you can specify the TPollSocketAbstract class to be used, if the default is not the one expected
- under Linux/POSIX, will set the open files maximum number for the current process to match
  the system hard limit: if your system has a low "ulimit -H -n" value, you may add the following
  line in your /etc/limits.conf or /etc/security/limits.conf file:
  * hard nofile 65535

destructor Destroy; override;

Finalize the sockets polling, and release all used memory

function GetOne(timeoutMS: integer; out notif: TPollSocketResult): boolean; virtual;
Retrieve the next pending notification, or let the poll wait for new
- if there is no pending notification, will poll and wait up to timeoutMS milliseconds for pending data
- returns true and set notif.events/tag with the corresponding notification
- returns false if no pending event was handled within the timeoutMS period
  this method is thread-safe, and could be called from several threads

function GetOneWithinPending(out notif: TPollSocketResult): boolean;
Retrieve the next pending notification
- returns true and set notif.events/tag with the corresponding notification
- returns false if no pending event is available
  this method is thread-safe, and could be called from several threads

function Subscribe(socket: TSocket; tag: TPollSocketTag; events: TPollSocketEvents): boolean; virtual;
Track modifications on one specified TSocket and tag
- the supplied tag value - maybe a PtrInt(aObject) - will be part of GetOne method results
  will create as many TPollSocketAbstract instances as needed, depending on the MaxSockets
  capability of the actual implementation class
  this method is thread-safe

function Unsubscribe(socket: TSocket; tag: TPollSocketTag): boolean; virtual;
Stop status modifications tracking on one specified TSocket and tag
- the socket should have been monitored by a previous call to Subscribe()
  this method is thread-safe

procedure Terminate;
Notify any GetOne waiting method to stop its polling loop

property Count: integer read fCount;
How many sockets are currently tracked

property PollClass: TPollSocketClass read fPollClass;
The actual polling class used to track socket state changes

property Terminated: boolean read fTerminated;
Set to true by the Terminate method
TPollSocketsSlot = object(TObject)
  Store information of one TPollAsynchSockets connection

  lastWSAError: integer;
  The last error reported by WSAGetLastError before the connection ends

  lockcounter: array[boolean] of integer;
  Lock/Unlock R/W thread acquisition (lighter than a TRTLCriticalSection)

  readbuf: SockString;
  The current read data buffer of this slot

  socket: TSocket;
  The associated TCP connection
  - equals 0 after TPollAsynchSockets.Stop

  writebuf: SockString;
  The current write data buffer of this slot

  function Lock(writer: boolean): boolean;
  Acquire an exclusive R/W access to this connection
  - returns true if slot has been acquired
  - returns false if it is used by another thread
  - warning: this method is not re-entrant

  function TryLock(writer: boolean; timeoutMS: cardinal): boolean;
  Try to acquire an exclusive R/W access to this connection
  - returns true if slot has been acquired
  - returns false if it is used by another thread, after the timeoutMS period
  - warning: this method is not re-entrant

  procedure UnLock(writer: boolean);
  Release exclusive R/W access to this connection

TPollAsynchSockets = class(TObject)
  Read/write buffer-oriented process of multiple non-blocking connections
  - to be used e.g. for stream protocols (e.g. WebSockets or IoT communication)
  - assigned sockets will be set in non-blocking mode, so that polling will work as expected: you
  should then never use directly the socket (e.g. via blocking TCrtSocket), but rely on this class for
  asynchronous process: OnRead() overriden method will receive all incoming data from input
  buffer, and Write() should be called to add some data to asynchronous output buffer
  - connections are identified as TObject instances, which should hold a TPollSocketsSlot record as
  private values for the polling process
  - ProcessRead/ProcessWrite methods are to be run for actual communication: either you call
  those methods from multiple threads, or you run them in loop from a single thread, then define a
  TSynThreadPool for running any blocking process (e.g. computing requests answers) from OnRead
  callbacks
  - inherited classes should override abstract OnRead, OnClose, OnError and SlotFromConnection
  methods according to the actual connection class
constructor Create; virtual;
    Initialize the read/write sockets polling
    - fRead and fWrite TpollSocketsBuffer instances will track pseRead or pseWrite events, and
      maintain input and output data buffers

destructor Destroy; override;
    Finalize buffer-oriented sockets polling, and release all used memory

function Start(connection: TObject): boolean; virtual;
    Assign a new connection to the internal poll
    - the TSocket handle will be retrieved via SlotFromConnection, and set in non-blocking mode
      from now on - it is not recommended to access it directly any more, but use Write() and handle
      OnRead() callback
    - fRead will poll incoming packets, then call OnRead to handle them, or Unsubscribe and delete
      the socket when pseClosed is notified
    - fWrite will poll for outgoing packets as specified by Write(), then send any pending data once
      the socket is ready

function Stop(connection: TObject): boolean; virtual;
    Remove a connection from the internal poll, and shutdown its socket
    - most of the time, the connection is released by OnClose when the other end shutdown the
      socket; but you can explicitly call this method when the connection (and its socket) is to be
      shutdown
    - this method won't call OnClose, since it is initiated by the class

function Write(connection: TObject; const data; datalen: integer; timeout: integer=5000): boolean; virtual;
    Add some data to the asynchronous output buffer of a given connection
    - this method may block if the connection is currently writing from another thread (which is not
      possible from TpollAsynchSockets.Write), up to timeout milliseconds

function WriteString(connection: TObject; const data: SockString): boolean;
    Add some data to the asynchronous output buffer of a given connection

procedure ProcessRead(timeoutMS: integer);
    One or several threads should execute this method
    - thread-safe handle of any incoming packets
    - if this method is called from a single thread, you should use a TSynThreadPool for any blocking
      process of OnRead events
    - otherwise, this method is thread-safe, and incoming packets may be consumed from a set of
      threads, and call OnRead with newly received data

procedure ProcessWrite(timeoutMS: integer);
    One or several threads should execute this method
    - thread-safe handle of any outgoing packets

procedure Terminate(waitforMS: integer);
    Notify internal socket polls to stop their polling loop ASAP

property Count: integer read GetCount;
    How many connections are currently managed by this instance
property Options: TPollAsynchSocketsOptions read fOptions write fOptions;
    Some processing options

property PollRead: TPollSockets read fRead;
    Low-level access to the polling class used for incoming data

property PollWrite: TPollSockets write fWrite;
    Low-level access to the polling class used for outgoing data

property ReadBytes: Int64 read fReadBytes;
    How many data bytes have been received by this instance

property ReadCount: integer read fReadCount;
    How many times data has been received by this instance

property WriteBytes: Int64 read fWriteBytes;
    How many data bytes have been sent by this instance

property WriteCount: integer read fWriteCount;
    How many times data has been sent by this instance

Types implemented in the SynCrtSock unit

PPollSocketsSlot = ^TPollSocketsSlot;
    Points to thread-safe information of one TPollAsynchSockets connection

PSockString = ^SockString;
    Points to a 8-bit raw storage variable, used for data buffer management

PtrInt = integer;
    FPC 64-bit compatibility integer type

SockString = type AnsiString;
    Define a 8-bit raw storage string type, used for data buffer management

SockUnicode = WideString;
    Define the fastest 16-bit Unicode string type of the compiler

TCrtSocketLayer = ( cs1TCP, cs1UDP, cs1UNIX );
    The available available network transport layer
    - either TCP/IP, UDP/IP or Unix sockets

TCrtSocketPending = ( cspSocketError, cspNoData, cspDataAvailable );
    Identify the incoming data availability in TCrtSocketSockReceivePending

THttpApiLogFields = set of ( hlfDate, hlfTime, hlfClientIP, hlfUserName, hlfSiteName, hlfComputerName, hlfServerIP, hlfMethod, hlfURIstem, hlfURIQuery, hlfStatus, hlfWIN32Status, hlfBytesSent, hlfBytesRecv, hlfTimeTaken, hlfServerPort, hlfUserAgent, hlfCookie, hlfReferer, hlfVersion, hlfHost, hlfSubStatus);
    Http.sys API 2.0 fields used for W3C logging
    - match low-level HTTP_LOG_FIELD_* constants as defined in HTTP 2.0 API

THttpApiLoggingFlags = set of ( hlfLocalTimeRollover, hlfUseUTF8Conversion, hlfLogErrorsOnly, hlfLogSuccessOnly);
    Http.sys API 2.0 logging option flags
    - used to alter the default logging behavior
- hlfLocalTimeRollover would force the log file rollovers by local time, instead of the default GMT time
- hlfUseUTF8Conversion will use UTF-8 instead of default local code page
- only one of hlfLogErrorsOnly and hlfLogSuccessOnly flag could be set at a time: if neither of them are present, both errors and success will be logged, otherwise mutually exclusive flags could be set to force only errors or success logging
- match low-level HTTP_LOGGING_FLAG_* constants as defined in HTTP 2.0 API

THttpApiLoggingRollOver = ( hlrSize, hlrDaily, hlrWeekly, hlrMonthly, hlrHourly );

Http.sys API 2.0 logging file rollover types
- match low-level HTTP_LOGGING_ROLLOVER_TYPE as defined in HTTP 2.0 API

THttpApiLoggingType = ( hltW3C, hltIIS, hltNCSA, hltRaw );

Http.sys API 2.0 logging file supported layouts
- match low-level HTTP_LOGGING_TYPE as defined in HTTP 2.0 API

THttpApiRequestAuthentications = set of ( haBasic, haDigest, haNtlm, haNegotiate, haKerberos);

Http.sys API 2.0 fields used for server-side authentication
- as used by THttpApiServer.SetAuthenticationSchemes/AuthenticationSchemes
- match low-level HTTP_AUTH_ENABLE_* constants as defined in HTTP 2.0 API

THttpApiWebSocketServerOnAcceptEvent = function(Ctxt: THttpServerRequest; var Conn: THttpApiWebSocketConnection): Boolean of object;

Event handlers for WebSocket

THttpClientSocketClass = class of THttpClientSocket;

Class-reference type (metaclass) of a HTTP client socket access
- may be either THttpClientSocket or THttpClientWebSockets (from SynBidirSock unit)

THttpRequestAuthentication = ( wraNone, wraBasic, wraDigest, wraNegotiate );

The supported authentication schemes which may be used by HTTP clients
- supported only by TWinHTTP class yet

THttpRequestClass = class of THttpRequest;

Store the actual class of a HTTP/1.1 client instance
- may be used to define at runtime which API to be used (e.g. WinHTTP, WinInet or LibCurl), following the Liskov substitution principle

THttpServerConnectionID = Int64;

A genuine identifier for a given client connection on server side
- maps http.sys ID, or is a genuine 31-bit value from increasing sequence

THttpServerConnectionIDDynArray = array of THttpServerConnectionID;

A dynamic array of client connection identifiers, e.g. for broadcasting

THttpServerRequestAuthentication = ( hraNone, hraFailed, hraBasic, hraDigest, hraNtlm, hraNegotiate, hraKerberos );

The server-side available authentication schemes
- as used by THttpServerRequest.AuthenticationStatus
- hraNone..hraKerberos will match low-level HTTP_REQUEST_AUTH_TYPE enum as defined in HTTP 2.0 API and

THttpServerRespClass = class of THttpServerResp;

Metaclass of HTTP response Thread
THttpServerSocketClass = class of THttpServerSocket;

Meta-class of the THttpServerSocket process
-usually used to override THttpServerSocket.GetRequest for instance

THttpServerSocketGetRequestResult = ( grError, grException, grOversizedPayload, grRejected, grTimeout, grHeaderReceived, grBodyReceived, grOwned );

Results of THttpServerSocket.GetRequest virtual method
- return grError if the socket was not connected any more, or grException if any exception occurred during the process
- grOversizedPayload is returned when MaximumAllowedContentLength is reached
- grRejected is returned when OnBeforeBody returned not 200
- grTimeout is returned when HeaderRetrieveAbortDelay is reached
- grHeaderReceived is returned for GetRequest({withbody=}false)
- grBodyReceived is returned for GetRequest({withbody=}true)
- grOwned indicates that this connection is now handled by another thread, e.g. asynchronous WebSockets

THttpSocketCompress = function(var DataRawByteString; Compress: boolean): AnsiString;

Event used to compress or uncompress some data during HTTP protocol
- should always return the protocol name for ACCEPT-ENCODING: header e.g. 'gzip' or 'deflate' for standard HTTP format, but you can add your own (like 'synlzo' or 'synlz')
- the data is compressed (if Compress=TRUE) or uncompressed (if Compress=FALSE) in the Data variable (i.e. it is modified in-place)
- to be used with THttpSocket.RegisterCompress method
- DataRawByteString type should be a generic AnsiString/RawByteString, which should be in practice a SockString or a RawByteString

THttpSocketCompressRecDynArray = array of THttpSocketCompressRec;

List of known compression algorithms

THttpSocketCompressSet = set of 0..31;

Identify some items in a list of known compression algorithms
- filled from ACCEPT-ENCODING: header value

TNotifyThreadEvent = procedure(Sender: TThread) of object;

Event prototype used e.g. by THttpServerGeneric.OnHttpThreadStart

TOnHttpServerAfterResponse = procedure(Ctxt: THttpServerRequest; const Code: cardinal) of object;

Event handler used by THttpServerGeneric.OnAfterResponse property
- Ctxt defines both input and output parameters
- Code defines the HTTP response code the (200 if OK, e.g.)

TOnHttpServerBeforeBody = function(const aURL,aMethod,aInHeaders, aInContentType,aRemoteIP: SockString; aContentLength: integer; aUseSSL: boolean): cardinal of object;

Event handler used by THttpServerGeneric.OnBeforeBody property
- if defined, is called just before the body is retrieved from the client
- supplied parameters reflect the current input state
- should return STATUS_SUCCESS=200 to continue the process, or an HTTP error code (e.g. STATUS_FORBIDDEN or STATUS_PAYLOADTOOLARGE) to reject the request

TOnHttpServerRequest = function(Cxt: THttpServerRequest): cardinal of object;

Event handler used by THttpServerGeneric.OnRequest property
- Ctx defines both input and output parameters
- result of the function is the HTTP error code (200 if OK, e.g.)
- OutCustomHeader will handle Content-Type/Location
- if OutContentType is HTTP_RESP_STATICFILE (i.e. 'STATICFILE' aka STATICFILE_CONTENT_TYPE in mORMot.pas), then OutContent is the UTF-8 file name of a file which must be sent directly to the client via http.sys or NGINX's X-Accel-Redirect; the OutCustomHeader should contain the proper 'Content-type: ....' value

```pascal
TOnHttpServerSendFile = function(Context: THttpServerRequest; const LocalFileName: TFileName): boolean of object;
```

Event handler used by THttpServer.Process to send a local file when HTTP_RESP_STATICFILE content-type is returned by the service
- can be defined e.g. to use NGINX X-Accel-Redirect header
- should return true if the Context has been modified to serve the file, or false so that the file will be manually read and sent from memory
- any exception during process will be returned as a STATUS_NOTFOUND page

```pascal
TPollAsynchSocketOnRead = ( sorContinue, sorClose );
```

Let TPollAsynchSockets.OnRead shutdown the socket if needed

```pascal
TPollAsynchSocketsOptions = set of (paoWritePollOnly);
```

Possible options for TPollAsynchSockets process
- by default, TPollAsynchSockets.Write will first try to send the data using Send() in non-blocking mode, unless paoWritePollOnly is defined, and fWrite will be used to poll output state and send it asynchronously

```pascal
TPollSocketClass = class of TPollSocketAbstract;
```

Meta-class of TPollSocketAbstract socket polling classes
- since TPollSocketAbstract.Create is declared as virtual, could be used to specify the proper polling class to add
- see PollSockClass function and TPollSocketAbstract.New method

```pascal
TPollSocketEvent = ( pseRead, pseWrite, pseError, pseClosed );
```

The events monitored by TPollSocketAbstract classes
- we don’t make any difference between urgent or normal read/write events

```pascal
TPollSocketEvents = set of TPollSocketEvent;
```

Set of events monitored by TPollSocketAbstract classes

```pascal
TPollSocketResults = array of TPollSocketResult;
```

All modifications returned by TPollSocketAbstract.WaitForModified

```pascal
TPollSocketTag = type Ptrlnt;
```

Some opaque value (which may be a pointer) associated with a polling event

```pascal
TSockStringDynArray = array of SockString;
```

Defines a dynamic array of SockString

```pascal
TWebSocketState = ( wsConnecting, wsOpen, wsClosing, wsClosedByClient, wsClosedByServer, wsClosedByGuard, wsClosedByShutdown );
```

Current state of a THttpApiWebSocketConnection

```pascal
TWinHttpDownload = function(Sender: TWinHttpAPI; CurrentSize, ContentLength, ChunkSize: DWORD; const ChunkData): boolean of object;
```

Event callback to process the download by chunks, not in memory
- used in TWinHttpAPI.OnDownload property
- CurrentSize is the current total number of downloaded bytes
- ContentLength is retrieved from HTTP headers, but may be 0 if not set
- ChunkSize is the size of the latest downloaded chunk, available in the untyped ChunkData memory buffer
- implementation should return TRUE to continue the download, or FALSE to abort the download process

\[ \text{TWinHttpProgress} = \text{procedure}(\text{Sender: TWinHttpAPI}; \text{CurrentSize}, \text{ContentLength}: \text{DWORD}) \]

of object;

\textbf{Event callback to track download progress, e.g. in the UI}
- used in TWinHttpAPI.OnProgress property
- CurrentSize is the current total number of downloaded bytes
- ContentLength is retrieved from HTTP headers, but may be 0 if not set

\[ \text{TWinHttpUpload} = \text{function}(\text{Sender: TWinHttpAPI}; \text{CurrentSize}, \text{ContentLength}: \text{DWORD}): \text{boolean} \]

of object;

\textbf{Event callback to track upload progress, e.g. in the UI}
- used in TWinHttpAPI.OnUpload property
- CurrentSize is the current total number of uploaded bytes
- ContentLength is the size of the content
- implementation should return TRUE to continue the upload, or FALSE to abort the upload process

\[ \text{WEB_SOCKET_BUFFER_TYPE} = \text{ULONG}; \]

\textit{The bit values used to construct the WebSocket frame header for httpapi.dll}
- not equals to WINHTTP_WEB_SOCKET_BUFFER_TYPE from winhttp.dll

\[ \text{WEB_SOCKET_CLOSE_STATUS} = \text{Word}; \]

\textit{WebSocket close status as defined by http://tools.ietf.org/html/rfc6455#section-7.4}

\[ \text{WEB_SOCKET_HANDLE} = \text{Pointer}; \]

\textit{Low-level API reference to a WebSocket session}

\[ \text{WINHTTP_WEB_SOCKET_BUFFER_TYPE} = \text{ULONG}; \]

\textit{Types of WebSocket buffers for winhttp.dll it is the different thing than WEB_SOCKET_BUFFER_TYPE for httpapi.dll}

\textbf{Constants implemented in the SynCrtSock unit}

\[ \text{HTTPAPI_AUTH_ENABLE_ALL} = [hraBasic..hraKerberos]; \]

\textit{Can be used with THttpApiServer.AuthenticationSchemes to enable all schemes}

\[ \text{HTTP_RESP_NORESPONSE} = '\text{!NORESPONSE}'; \]

\textit{Used to notify e.g. the THttpServerRequest not to wait for any response from the client}
- is not to be used in normal HTTP process, but may be used e.g. by TWebSocketProtocolRest.ProcessFrame() to avoid to wait for an incoming response from the other endpoint
- should match NORESPONSE\_CONTENT\_TYPE constant defined in mORMot.pas unit

\[ \text{HTTP_RESP_STATICFILE} = '\text{!STATICFILE}'; \]

\textit{Internal HTTP content-type for efficient static file sending}
- detected e.g. by http.sys' THttpApiServer.Request or via the NGINX X-Accel-Redirect header's THttpServer.Process (see THttpServer.NginxSendFileFrom) for direct sending with no local buffering
- the OutCustomHeader should contain the proper 'Content-type: ....' corresponding to the file (e.g.
by calling GetMimeContentType() function from SynCommons supplyings the file name
- should match HTML_CONTENT_STATICFILE constant defined in mORMot.pas unit

```pascal
SMTP_DEFAULT = 'user:password@smtpserver:port';

The layout of TSMTPConnection.FromText method
```

```pascal
STATUS_ACCEPTED = 202;
HTTP Status Code for "Accepted"

STATUS_BADREQUEST = 400;
HTTP Status Code for "Bad Request"

STATUS_CREATED = 201;
HTTP Status Code for "Created"

STATUS_FORBIDDEN = 403;
HTTP Status Code for "Forbidden"

STATUS_HTTPVERSIONNONSUPPORTED = 505;
HTTP Status Code for "HTTP Version Not Supported"

STATUS_NOCONTENT = 204;
HTTP Status Code for "No Content"

STATUS_NOTACCEPTABLE = 406;
HTTP Status Code for "Not Acceptable"

STATUS_NOTFOUND = 404;
HTTP Status Code for "Not Found"

STATUS_NOTIMPLEMENTED = 501;
HTTP Status Code for "Not Implemented"

STATUS_NOTMODIFIED = 304;
HTTP Status Code for "Not Modified"

STATUS_PARTIALCONTENT = 206;
HTTP Status Code for "Partial Content"

STATUS_PAYLOADTOOLARGE = 413;
HTTP Status Code for "Payload Too Large"

STATUS_SERVERERROR = 500;
HTTP Status Code for "Internal Server Error"

STATUS_SUCCESS = 200;
HTTP Status Code for "Success"

STATUS_UNAUTHORIZED = 401;
HTTP Status Code for "Unauthorized"

WEB_SOCKET_ABORTED_CLOSE_STATUS : WEB_SOCKET_CLOSE_STATUS = 1006;
The connection was closed without sending or receiving a close frame

WEB_SOCKET_BINARY_FRAGMENT_BUFFER_TYPE: WEB_SOCKET_BUFFER_TYPE = $80000003;
The buffer contains part of a binary message
<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEB_SOCKET_BINARY_MESSAGE_BUFFER_TYPE</td>
<td>The buffer contains the last, and possibly only, part of a binary message</td>
</tr>
<tr>
<td>WEB_SOCKET_CLOSE_BUFFER_TYPE</td>
<td>The buffer contains a close message</td>
</tr>
<tr>
<td>WEB_SOCKETEMPTY_CLOSE_STATUS</td>
<td>No close status code was provided</td>
</tr>
<tr>
<td>WEB_SOCKET_ENDPOINT_UNAVAILABLE_CLOSE_STATUS</td>
<td>The endpoint is going away and thus closing the connection</td>
</tr>
<tr>
<td>WEB_SOCKET_INVALID_DATA_TYPE_CLOSE_STATUS</td>
<td>The endpoint cannot receive this type of data</td>
</tr>
<tr>
<td>WEB_SOCKET_INVALID_PAYLOAD_CLOSE_STATUS</td>
<td>Data within a message is not consistent with the type of the message</td>
</tr>
<tr>
<td>WEB_SOCKET_MESSAGE_TOO_BIG_CLOSE_STATUS</td>
<td>The message sent was too large to process</td>
</tr>
<tr>
<td>WEB_SOCKET_PING_PONG_BUFFER_TYPE</td>
<td>The buffer contains a ping or pong message</td>
</tr>
<tr>
<td>WEB_SOCKET_POLICY_VIOLATION_CLOSE_STATUS</td>
<td>The message violates an endpoint's policy</td>
</tr>
<tr>
<td>WEB_SOCKET_PROTOCOL_ERROR_CLOSE_STATUS</td>
<td>Peer detected protocol error and it is closing the connection</td>
</tr>
<tr>
<td>WEB_SOCKET_SECURE_HANDSHAKE_ERROR_CLOSE_STATUS</td>
<td>The TLS handshake could not be completed</td>
</tr>
<tr>
<td>WEB_SOCKET_SERVER_ERROR_CLOSE_STATUS</td>
<td>An unexpected condition prevented the server from fulfilling the request</td>
</tr>
<tr>
<td>WEB_SOCKET_SUCCESS_CLOSE_STATUS</td>
<td>Close completed successfully</td>
</tr>
<tr>
<td>WEB_SOCKET_UNSOLICITED_PONG_BUFFER_TYPE</td>
<td>The buffer contains an unsolicited pong message</td>
</tr>
<tr>
<td>WEB_SOCKET_UNSUPPORTED_EXTENSIONS_CLOSE_STATUS</td>
<td>A client endpoint expected the server to negotiate one or more extensions, but the server didn't return them in the response message of the WebSocket handshake</td>
</tr>
<tr>
<td>WEB_SOCKET_UTF8_FRAGMENT_BUFFER_TYPE</td>
<td>The buffer contains part of a UTF8 message</td>
</tr>
<tr>
<td>WEB_SOCKET_UTF8_MESSAGE_BUFFER_TYPE</td>
<td>The buffer contains the last, and possibly only, part of a UTF8 message</td>
</tr>
</tbody>
</table>
XPOWEREDOS = 'Windows' ;

The running Operating System

XPOWEREDPROGRAM = 'mORMot 1.18';

The full text of the current Synopse mORMot framework version
- match the value defined in SynCommons.pas and SynopseCommit.inc
- we don't supply full version number with build revision, to reduce potential attack surface

### Functions or procedures implemented in the SynCrtSock unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsynchRecv</td>
<td>Low-level direct call of the socket recv() function</td>
<td>1134</td>
</tr>
<tr>
<td>AsynchSend</td>
<td>Low-level direct call of the socket send() function</td>
<td>1134</td>
</tr>
<tr>
<td>AsynchSocket</td>
<td>Low-level change of a socket to be in non-blocking mode</td>
<td>1134</td>
</tr>
<tr>
<td>AuthorizationBearer</td>
<td>Compute the 'Authorization: Bearer ####' HTTP header of a given token</td>
<td>1135</td>
</tr>
<tr>
<td>CallServer</td>
<td>Low-level direct creation of a TSocket handle for TCP, UDP or UNIX layers</td>
<td>1135</td>
</tr>
<tr>
<td>DirectShutdown</td>
<td>Low-level direct shutdown of a given socket</td>
<td>1135</td>
</tr>
<tr>
<td>GetIPAddresses</td>
<td>Enumerate all IP addresses of the current computer</td>
<td>1135</td>
</tr>
<tr>
<td>GetIPAddressesText</td>
<td>Returns all IP addresses of the current computer as a single CSV text</td>
<td>1135</td>
</tr>
<tr>
<td>GetMacAddresses</td>
<td>Enumerate all Mac addresses of the current computer</td>
<td>1135</td>
</tr>
<tr>
<td>GetMacAddressesText</td>
<td>Enumerate all Mac addresses of the current computer as 'name1=addr1 name2=addr2'</td>
<td>1135</td>
</tr>
<tr>
<td>GetRemoteIP</td>
<td>Retrieve the text-converted remote IP address of a client socket</td>
<td>1135</td>
</tr>
<tr>
<td>GetRemoteMacAddress</td>
<td>Remotly get the MAC address of a computer, from its IP Address</td>
<td>1135</td>
</tr>
<tr>
<td>HtmlEncode</td>
<td>Escaping of HTML codes like &lt; &gt; &amp; &quot;</td>
<td>1135</td>
</tr>
<tr>
<td>HttpChunkToHex32</td>
<td>Decode a HTTP chunk length</td>
<td>1135</td>
</tr>
<tr>
<td>HttpGet</td>
<td>Retrieve the content of a web page, using the HTTP/1.1 protocol and GET method</td>
<td>1136</td>
</tr>
<tr>
<td>HttpGet</td>
<td>Retrieve the content of a web page, using the HTTP/1.1 protocol and GET method</td>
<td>1136</td>
</tr>
<tr>
<td>HttpGet</td>
<td>Retrieve the content of a web page, using the HTTP/1.1 protocol and GET method</td>
<td>1136</td>
</tr>
<tr>
<td>HttpGetAuth</td>
<td>Retrieve the content of a web page, using HTTP/1.1 GET method and a token</td>
<td>1136</td>
</tr>
<tr>
<td>HttpPost</td>
<td>Send some data to a remote web server, using the HTTP/1.1 protocol and POST method</td>
<td>1136</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>IP4Text</td>
<td>Compute the '1.2.3.4' text representation of a raw IP4 binary</td>
<td>1136</td>
</tr>
<tr>
<td>IPText</td>
<td>Compute the text representation of a IP4/IP6 low-level connection</td>
<td>1136</td>
</tr>
<tr>
<td>MainHttpClass</td>
<td>Returns the best THttpRequest class, depending on the system it runs on</td>
<td>1136</td>
</tr>
<tr>
<td>Open</td>
<td>Create a TCrtSocket, returning nil on error (useful to easily catch socket error exception ECrtSocket)</td>
<td>1136</td>
</tr>
<tr>
<td>OpenHttp</td>
<td>Create a THttpClientSocket, returning nil on error</td>
<td>1136</td>
</tr>
<tr>
<td>PollSocketClass</td>
<td>Returns the TPollSocketAbstract class best fitting with the current Operating System</td>
<td>1136</td>
</tr>
<tr>
<td>ReplaceMainHttpClass</td>
<td>Low-level forcing of another THttpRequest class</td>
<td>1136</td>
</tr>
<tr>
<td>ResolveName</td>
<td>Retrieve the IP address from a computer name</td>
<td>1136</td>
</tr>
<tr>
<td>SendEmail</td>
<td>Send an email using the SMTP protocol</td>
<td>1137</td>
</tr>
<tr>
<td>SendEmailSubject</td>
<td>Convert a supplied subject text into an Unicode encoding</td>
<td>1137</td>
</tr>
<tr>
<td>SockBase64Decode</td>
<td>Base64 decoding of a string</td>
<td>1137</td>
</tr>
<tr>
<td>SockBase64Encode</td>
<td>Base64 encoding of a string</td>
<td>1137</td>
</tr>
<tr>
<td>SocketErrorMessage</td>
<td>Low-level text description of Socket error code</td>
<td>1137</td>
</tr>
<tr>
<td>StatusCodeToReason</td>
<td>Retrieve the HTTP reason text from a code</td>
<td>1137</td>
</tr>
<tr>
<td>WinHTTP_WebSocketEnabled</td>
<td>Is HTTP.SYS web socket API available on the target system Windows 8 and UP</td>
<td>1137</td>
</tr>
</tbody>
</table>

**function** AsynchRecv(sock: TSocket; buf: pointer; buflen: integer): integer;

*Low-level direct call of the socket recv() function*

- by-pass overridden blocking recv() e.g. in SynFPCSock, so will work if the socket is in non-blocking mode, as with AsynchSocket/TPollAsynchSockets

**function** AsynchSend(sock: TSocket; buf: pointer; buflen: integer): integer;

*Low-level direct call of the socket send() function*

- by-pass overridden blocking send() e.g. in SynFPCSock, so will work if the socket is in non-blocking mode, as with AsynchSocket/TPollAsynchSockets

**function** AsynchSocket(sock: TSocket): boolean;

*Low-level change of a socket to be in non-blocking mode*

- used e.g. by TPollAsynchSockets.Start
**function AuthorizationBearer(const AuthToken: SockString): SockString;**

*Compute the 'Authorization: Bearer ####' HTTP header of a given token value*

**function CallServer(const Server, Port: SockString; doBind: boolean; aLayer: TCrtSocketLayer; ConnectTimeout: DWORD): TSocket;**

*Low-level direct creation of a TSocket handle for TCP, UDP or UNIX layers*
- doBind=true will call Bind() to create a server socket instance
- doBind=false will call Connect() to create a client socket instance

**procedure DirectShutdown(sock: TSocket; rdwr: boolean=false);**

*Low-level direct shutdown of a given socket*

**function GetIPAddresses(Kind: TIPAddress = tiaAny): TSockStringDynArray;**

*Enumerate all IP addresses of the current computer*
- may be used to enumerate all adapters

**function GetIPAddressesText(const Sep: SockString = ' '; PublicOnly: boolean = false): SockString;**

*Returns all IP addresses of the current computer as a single CSV text*
- may be used to enumerate all adapters

**function GetMacAddresses: TMacAddressDynArray;**

*Enumerate all Mac addresses of the current computer*

**function GetMacAddressesText: SockString;**

*Enumerate all Mac addresses of the current computer as 'name1=addr1 name2=addr2'*

**function GetRemoteIP(aClientSock: TSocket): SockString;**

*Retrieve the text-converted remote IP address of a client socket*

**function GetRemoteMacAddress(const IP: SockString): SockString;**

*Remotely get the MAC address of a computer, from its IP Address*
- only works under Win2K and later
- return the MAC address as a 12 hexa chars ('0050C204C80A' e.g.)

**function HtmlEncode(const s: SockString): SockString;**

*Escaping of HTML codes like < > & "*

**function HttpChunkToHex32(p: PAnsiChar): integer;**

*Decode a HTTP chunk length*

**function HttpGet(const aURI: SockString; outHeaders: PSockString=nil; forceNotSocket: boolean=false; outStatus: PInteger=nil): SockString; overload;**

*Retrieve the content of a web page, using the HTTP/1.1 protocol and GET method*
- this method will use a low-level THttpClientSock socket for plain http URI, or TWinHTTP/TCurlHTTP for any https URI, or if forceNotSocket is set to true

**function HttpGet(const server, port: SockString; const url1: SockString; const inHeaders: SockString; outHeaders: PSockString=nil; aLayer: TCrtSocketLayer = cs1TCP): SockString; overload;**

*Retrieve the content of a web page, using the HTTP/1.1 protocol and GET method*
- this method will use a low-level THttpClientSock socket: if you want something able to use your computer proxy, take a look at TWinINet.Get()
**function** `HttpGet(const aURI: SockString; const inHeaders: SockString; outHeaders: PSockString=nil; forceNotSocket: boolean=false; outStatus: PInteger=nil): SockString; overload;`

*Retrieve the content of a web page, using the HTTP/1.1 protocol and GET method*
- this method will use a low-level THttpClientSock socket for plain http URI, or TWinHTTP/TCurlHTTP for any https URI

**function** `HttpGetAuth(const aURI, aAuthToken: SockString; outHeaders: PSockString=nil; forceNotSocket: boolean=false; outStatus: PInteger=nil): SockString; overload;`

*Retrieve the content of a web page, using HTTP/1.1 GET method and a token*
- this method will use a low-level THttpClientSock socket and its GetAuth method
- if AuthToken<>'', will add an header with 'Authorization: Bearer '+AuthToken

**function** `HttpPost(const server, port: SockString; const url, Data, DataType: SockString; outData: PSockString=nil; const auth: SockString=''): boolean;`

*Send some data to a remote web server, using the HTTP/1.1 protocol and POST method*

**procedure** `IP4Text(const ip4addr; var result: SockString); overload;`

*Compute the '1.2.3.4' text representation of a raw IP4 binary*

**procedure** `IPText(const sin: TVarSin; var result: SockString; localasvoid: boolean=false);`

*Compute the text representation of a IP4/IP6 low-level connection*

**function** `MainHttpClass: THttpRequestClass;`

*Returns the best THttpRequest class, depending on the system it runs on*
- e.g. TWinHTTP or TCurlHTTP
- consider using TSimpleHttpClient if you just need a simple connection

**function** `Open(const aServer, aPort: SockString; aTLS: boolean=false): TCrtSocket;`

*Create a TCrtSocket, returning nil on error (useful to easily catch socket error exception ECrtSocket)*

**function** `OpenHttp(const aServer, aPort: SockString; aTLS: boolean=false; aLayer: TCrtSocketLayer = cslTCP): THttpClientSocket; overload;`

*Create a THttpClientSocket, returning nil on error*
- useful to easily catch socket error exception ECrtSocket

**function** `OpenHttp(const aURI: SockString; aAddress: PSockString=nil): THttpClientSocket; overload;`

*Create a THttpClientSocket, returning nil on error*
- useful to easily catch socket error exception ECrtSocket

**function** `PollSocketClass: TPollSocketClass;`

*Returns the TPollSocketAbstract class best fitting with the current Operating System*
- as used by TPollSocketAbstract.New method

**procedure** `ReplaceMainHttpClass(aClass: THttpRequestClass);`

*Low-level forcing of another THttpRequest class*
- could be used if we found out that the current MainHttpClass failed (which could easily happen with TCurlHTTP if the library is missing or deprecated)

**function** `ResolveName(const Name: SockString; Family: Integer=AF_INET; SockProtocol: Integer=IPPROTO_TCP; SockType: integer=SOCK_STREAM): SockString;`

*Retrieve the IP address from a computer name*

Send an email using the SMTP protocol
- retry true on success
- the Subject is expected to be in plain 7 bit ASCII, so you could use SendEmailSubject() to encode it as Unicode, if needed
- you can optionally set the encoding charset to be used for the Text body

function SendEmail(const Server: TSMTPConnection; const From, CSVDest, Subject, Text: SockString; const Headers: SockString=''; const TextCharSet: SockString = 'ISO-8859-1'; aTLS: boolean=false): boolean; overload;

Send an email using the SMTP protocol
- retry true on success
- the Subject is expected to be in plain 7 bit ASCII, so you could use SendEmailSubject() to encode it as Unicode, if needed
- you can optionally set the encoding charset to be used for the Text body

function SendEmailSubject(const Text: string): SockString;

Convert a supplied subject text into an Unicode encoding
- will convert the text into UTF-8 and append '='UTF-8?B'
- for pre-Unicode versions of Delphi, Text is expected to be already UTF-8 encoded - since Delphi 2010, it will be converted from UnicodeString

function SockBase64Decode(const s: SockString): SockString;

Base64 decoding of a string
- consider using more efficient Base64ToBin() from SynCommons.pas instead

function SockBase64Encode(const s: SockString): SockString;

Base64 encoding of a string
- used internally for STMP email sending
- consider using more efficient BinToBase64() from SynCommons.pas instead

function SocketErrorMessage(Error: integer=-1): string;

Low-level text description of Socket error code
- if Error is -1, will call WSAGetLastError to retrieve the last error code

function StatusCodeToReason(Code: cardinal): SockString;

Retrieve the HTTP reason text from a code
- e.g. StatusCodeToReason(200)='OK'
- see http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
- mORMot.StatusCodeToErrorMsg() will call this function

function WinHTTP_WebSocketEnabled: boolean;

Is HTTP.SYS web socket API available on the target system Windows 8 and UP

Variables implemented in the SynCrtSock unit
DefaultListenBacklog: integer = SOMAXCONN;

Queue length for completely established sockets waiting to be accepted, a backlog parameter for listen() function. If queue overflows client got ECONNREFUSED error for connect() call
- for windows default is taken from SynWinSock ($7fffffff) and should not be modified. Actual limit is 200;
- for Unix default is taken from SynFPCSock (128 as in linux kernel >2.2), but actual value is min(DefaultListenBacklog, /proc/sys/net/core/somaxconn)

HTTP_DEFAULT_CONNECTTIMEOUT: integer = 60000;

THttpRequest timeout default value for remote connection
- default is 60 seconds
- used e.g. by THttpRequest, TSQLHttpClientRequest and TSQLHttpClientGeneric

HTTP_DEFAULT_RECEIVETIMEOUT: integer = 30000;

THttpRequest timeout default value for data receiving
- default is 30 seconds
- used e.g. by THttpRequest, TSQLHttpClientRequest and TSQLHttpClientGeneric
- you can override this value by setting the corresponding parameter in THttpRequest.Create() constructor

HTTP_DEFAULT_RESOLVETIMEOUT: integer = 0;

THttpRequest timeout default value for DNS resolution
- leaving to 0 will let system default value be used

HTTP_DEFAULT_SENDTIMEOUT: integer = 30000;

THttpRequest timeout default value for data sending
- default is 30 seconds
- used e.g. by THttpRequest, TSQLHttpClientRequest and TSQLHttpClientGeneric
- you can override this value by setting the corresponding parameter in THttpRequest.Create() constructor

RemoteIPLocalHostAsVoidInServers: boolean = true;

Defines if a connection from the loopback should be reported as ” (no Remote-IP - which is the default) or as ‘127.0.0.1’ (force to false)
- used by both TCrtSock.AcceptRequest and THttpApiServer.Execute servers
27.7. SynCrypto.pas unit

Purpose: Fast cryptographic routines (hashing and cypher)
- implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT
- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the SynCrypto unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLZ</td>
<td>SynLZ Compression routines</td>
<td>1393</td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>
Objects implemented in the SynCrypto unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJWTException</td>
<td>Exception raised when running JSON Web Tokens</td>
<td>1176</td>
</tr>
<tr>
<td>ESynCrypto</td>
<td>Class of Exceptions raised by this unit</td>
<td>1143</td>
</tr>
<tr>
<td>IProtocol</td>
<td>Perform safe communication after unilateral or mutual authentication</td>
<td>1175</td>
</tr>
<tr>
<td>TAES</td>
<td>Handle AES cypher/uncypher</td>
<td>1144</td>
</tr>
<tr>
<td>TAESAbstract</td>
<td>Handle AES cypher/uncypher with chaining</td>
<td>1146</td>
</tr>
<tr>
<td>TAESAbstractAEAD</td>
<td>AEAD (authenticated-encryption with associated-data) abstract class</td>
<td>1155</td>
</tr>
<tr>
<td>TAESAbstractEncryptOnly</td>
<td>Abstract parent class for chaining modes using only AES encryption</td>
<td>1153</td>
</tr>
<tr>
<td>TAESAbstractSyn</td>
<td>Handle AES cypher/uncypher with chaining with out own optimized code</td>
<td>1152</td>
</tr>
<tr>
<td>TAESAbstract_API</td>
<td>Handle AES cypher/uncypher using Windows CryptoAPI and the official</td>
<td>1157</td>
</tr>
<tr>
<td></td>
<td>Microsoft AES Cryptographic Provider (PROV_RSA_AES)</td>
<td></td>
</tr>
<tr>
<td>TAESCBC</td>
<td>Handle AES cypher/uncypher with Cipher-block chaining (CBC)</td>
<td>1153</td>
</tr>
<tr>
<td>TAESCBC_API</td>
<td>Handle AES cypher/uncypher Cipher-block chaining (CBC) using Windows</td>
<td>1158</td>
</tr>
<tr>
<td></td>
<td>CryptoAPI</td>
<td></td>
</tr>
<tr>
<td>TAESCFB</td>
<td>Handle AES cypher/uncypher with Cipher feedback (CFB)</td>
<td>1153</td>
</tr>
<tr>
<td>TAESCFBCRC</td>
<td>AEAD combination of AES with Cipher feedback (CFB) and 256-bit MAC</td>
<td>1155</td>
</tr>
<tr>
<td>TAESCFB_API</td>
<td>Handle AES cypher/uncypher Cipher feedback (CFB) using Windows CryptoAPI</td>
<td>1158</td>
</tr>
<tr>
<td>TAESCCTR</td>
<td>Handle AES cypher/uncypher with 64-bit Counter mode (CTR)</td>
<td>1154</td>
</tr>
<tr>
<td>TAESECB</td>
<td>Handle AES cypher/uncypher without chaining (ECB)</td>
<td>1152</td>
</tr>
<tr>
<td>TAESECB_API</td>
<td>Handle AES cypher/uncypher without chaining (ECB) using Windows CryptoAPI</td>
<td>1157</td>
</tr>
<tr>
<td>TAESFull</td>
<td>AES and XOR encryption object for easy direct memory or stream access</td>
<td>1168</td>
</tr>
<tr>
<td>TAESFullHeader</td>
<td>Internal header for storing our AES data with salt and CRC</td>
<td>1168</td>
</tr>
<tr>
<td>TAESGCM</td>
<td>Handle AES-GCM cypher/uncypher with built-in authentication</td>
<td>1156</td>
</tr>
<tr>
<td>TAESGCMEngine</td>
<td>Low-level AES-GCM processing</td>
<td>1145</td>
</tr>
<tr>
<td>TAESIVCTR</td>
<td>Used internally by TAESAbstract to detect replay attacks</td>
<td>1146</td>
</tr>
<tr>
<td>TAESLocked</td>
<td>Thread-safe class containing a TAES encryption/decryption engine</td>
<td>1158</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TAESMAC256</td>
<td>Internal 256-bit structure used for TAESAbstractAEAD MAC storage</td>
<td>1154</td>
</tr>
<tr>
<td>TAESOFB</td>
<td>Handle AES cypher/uncypher with Output feedback (OFB)</td>
<td>1153</td>
</tr>
<tr>
<td>TAESOFBCRC</td>
<td>AEAD combination of AES with Output feedback (OFB) and 256-bit MAC</td>
<td>1156</td>
</tr>
<tr>
<td>TAESOFB_API</td>
<td>Handle AES cypher/uncypher Output feedback (OFB) using Windows CryptoAPI</td>
<td>1158</td>
</tr>
<tr>
<td>TAESPRNG</td>
<td>Cryptographic pseudorandom number generator (CSPRNG) based on AES-256</td>
<td>1158</td>
</tr>
<tr>
<td>TAESPRNGSystem</td>
<td>TAESPRNG-compatible class using Operating System pseudorandom source</td>
<td>1161</td>
</tr>
<tr>
<td>TAESWriteStream</td>
<td>AES encryption stream</td>
<td>1169</td>
</tr>
<tr>
<td>THash128History</td>
<td>Stores an array of THash128 to check for their unicity</td>
<td>1143</td>
</tr>
<tr>
<td>THMAC_CRC32C</td>
<td>Compute the HMAC message authentication code using crc32c as hash function</td>
<td>1174</td>
</tr>
<tr>
<td>THMAC_SHA1</td>
<td>Compute the HMAC message authentication code using SHA-1 as hash function</td>
<td>1170</td>
</tr>
<tr>
<td>THMAC_SHA256</td>
<td>Compute the HMAC message authentication code using SHA-256 as hash function</td>
<td>1171</td>
</tr>
<tr>
<td>THMAC_SHA384</td>
<td>Compute the HMAC message authentication code using SHA-384 as hash function</td>
<td>1170</td>
</tr>
<tr>
<td>THMAC_SHA512</td>
<td>Compute the HMAC message authentication code using SHA-512 as hash function</td>
<td>1171</td>
</tr>
<tr>
<td>TJWTAbstract</td>
<td>Abstract parent class for implementing JSON Web Tokens</td>
<td>1177</td>
</tr>
<tr>
<td>TJWTContent</td>
<td>JWT decoded content, as processed by TJWTAbstract</td>
<td>1176</td>
</tr>
<tr>
<td>TJWTHS256</td>
<td>Implements JSON Web Tokens using 'HS256' (HMAC SHA-256) algorithm</td>
<td>1181</td>
</tr>
<tr>
<td>TJWTHS384</td>
<td>Implements JSON Web Tokens using 'HS384' (HMAC SHA-384) algorithm</td>
<td>1181</td>
</tr>
<tr>
<td>TJWTHS512</td>
<td>Implements JSON Web Tokens using 'HS512' (HMAC SHA-512) algorithm</td>
<td>1181</td>
</tr>
<tr>
<td>TJWTNone</td>
<td>Implements JSON Web Tokens using 'none' algorithm</td>
<td>1180</td>
</tr>
<tr>
<td>TJWT53224</td>
<td>Experimental JSON Web Tokens using SHA3-224 algorithm</td>
<td>1181</td>
</tr>
<tr>
<td>TJWT53256</td>
<td>Experimental JSON Web Tokens using SHA3-256 algorithm</td>
<td>1181</td>
</tr>
<tr>
<td>TJWT53384</td>
<td>Experimental JSON Web Tokens using SHA3-384 algorithm</td>
<td>1181</td>
</tr>
<tr>
<td>TJWT53512</td>
<td>Experimental JSON Web Tokens using SHA3-512 algorithm</td>
<td>1181</td>
</tr>
<tr>
<td>TJWT535128</td>
<td>Experimental JSON Web Tokens using SHA3-SHAKE128 algorithm</td>
<td>1182</td>
</tr>
</tbody>
</table>
Objects | Description | Page
---|---|---
TJWTS35256 | Experimental JSON Web Tokens using SHA3-SHAKE256 algorithm | 1182
TJWTSynSignerAbstract | Abstract parent of JSON Web Tokens using HMAC-SHA2 or SHA-3 algorithms | 1180
TMD5 | Handle MD5 hashing | 1167
TProtocolAES | Implements a secure protocol using AES encryption | 1176
TProtocolNone | Implements a fake no-encryption protocol | 1175
TRC4 | Handle RC4 encryption/decryption | 1167
TSHA1 | Handle SHA-1 hashing | 1162
TSHA256 | Handle SHA-256 hashing | 1162
TSHA3 | Handle SHA-3 (Keccak) hashing | 1164
TSHA384 | Handle SHA-384 hashing | 1163
TSHA512 | Handle SHA-512 hashing | 1164
TSynHasher | Convenient multi-algorithm hashing wrapper | 1174
TSynSigner | A generic wrapper object to handle digital HMAC-SHA-2/SHA-3 signatures | 1172
TSynSignerParams | JSON-serialization ready object as used by TSynSigner.PBKDF2 overloaded methods | 1172

ESynCrypto = class(ESynException)

Class of Exceptions raised by this unit

THash128History = object(TObject)

Stores an array of THash128 to check for their unicity
- used e.g. to implement TAESAbstract.IVHistoryDepth property, but may be also used to efficiently store a list of 128-bit IPv6 addresses

Count: integer;
How many THash128 values are currently stored

Depth: integer;
How many THash128 values can be stored

function Add(const hash: THash128): boolean;
Add a hash value to the stored entries, checking for duplicates
- returns true if the hash was added, or false if it did already appear

function Exists(const hash: THash128): boolean;
O(n) fast search of a hash value in the stored entries
- returns true if the hash was found, or false if it did not appear
procedure Init(size, maxsize: integer);

  *Initialize the storage for a given history depth*
  
  - if Count reaches Depth, then older items will be removed

**TAES = object(TObject)**

*Handle AES cypher/uncipher*

- this is the default Electronic codebook (ECB) mode
- this class will use AES-NI hardware instructions, if available
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one initialized instance (warning: not for Padlock)

function DecryptInit(const Key; KeySize: cardinal): boolean;

  *Initialize AES contexts for uncypher*
  
  - first method to call before using this object for decryption
  - KeySize is in bits, i.e. 128,192,256

function DecryptInitFrom(const Encryption: TAES; const Key; KeySize: cardinal): boolean;

  *Initialize AES contexts from another TAES.EncryptInit*

function DoInit(const Key; KeySize: cardinal; doEncrypt: boolean): boolean;

  *Generic initialization method for AES contexts*
  
  - call either EncryptInit() either DecryptInit() method

function EncryptInit(const Key; KeySize: cardinal): boolean;

  *Initialize AES contexts for cypher*
  
  - first method to call before using this object for encryption
  - KeySize is in bits, i.e. 128,192,256

function Initialized: boolean;

  *TRUE if the context was initialized via EncryptInit/DecryptInit*

function KeyBits: integer;

  *Returns the key size in bits (128/192/256)*

function UsesAESNI: boolean;

  *Return TRUE if the AES-NI instruction sets are available on this CPU*

procedure Decrypt(var B: TAESBlock); overload;

  *Decrypt an AES data block*

procedure Decrypt(const BI: TAESBlock; var BO: TAESBlock); overload;

  *Decrypt an AES data block into another data block*

procedure DoBlocks(pIn, pOut: PAESBlock; Count: integer; doEncrypt: boolean); overload;

  *Perform the AES cypher or uncypher to continuous memory blocks*
  
  - call either Encrypt() either Decrypt() method

procedure DoBlocks(pIn, pOut: PAESBlock; out oIn, oOut: PAESBlock; Count: integer; doEncrypt: boolean); overload;

  *Perform the AES cypher or uncypher to continuous memory blocks*
  
  - call either Encrypt() either Decrypt() method
**procedure** DoBlocksOFB(const iv: TAESBlock; src, dst: pointer; blockcount: PtrUInt);

Performs AES-OFB encryption and decryption on whole blocks
- may be called instead of TAESOFB when only a raw TAES is available
- this method is thread-safe (except if padlock is used)

**procedure** DoBlocksThread(var bIn, bOut: PAESBlock; Count: integer; doEncrypt: boolean);

Perform the AES cypher or uncypher to continuous memory blocks
- this special method will use Threads for bigs blocks (>512KB) if multi-CPU
- call either Encrypt() either Decrypt() method

**procedure** Done;

Finalize AES contexts for both cypher and uncypher
- would fill the TAES instance with zeros, for safety
- is only mandatory when padlock is used

**procedure** Encrypt(const BI: TAESBlock; var BO: TAESBlock); overload;

Encrypt an AES data block into another data block

**procedure** Encrypt(var B: TAESBlock); overload;

Encrypt an AES data block

---

TAESGCMEngine = object(TObject)

Low-level AES-GCM processing
- implements standard AEAD (authenticated-encryption with associated-data) algorithm, as defined by NIST and

**function** Add_AAD(pAAD: pointer; aLen: PtrInt): boolean;

Append some data to be authenticated, but not encrypted

**function** Decrypt(ctp, ptp: Pointer; ILen: PtrInt; ptag: pointer=nil; tlen: PtrInt=0): boolean;

Decrypt a buffer with AES-GCM, updating the associated authentication data
- also validate the GMAC with the supplied ptag/tlen if ptag<>nil, and skip the AES-CTR phase if the authentication doesn't match

**function** Encrypt(ptp, ctp: Pointer; ILen: PtrInt): boolean;

Encrypt a buffer with AES-GCM, updating the associated authentication data

**function** Final(out tag: TAESBlock; andDone: boolean=true): boolean;

Finalize the AES-GCM encryption, returning the authentication tag
- will also flush the AES context to avoid forensic issues, unless andDone is forced to false

**function** FullDecryptAndVerify(const Key; KeyBits: PtrInt; pIV: pointer; IV_len: PtrInt; pAAD: pointer; aLen: PtrInt; ctp, ptp: Pointer; pLen: PtrInt; ptag: pointer; tLen: PtrInt): boolean;

Single call AES-GCM decryption and verification process

**function** FullEncryptAndAuthenticate(const Key; KeyBits: PtrInt; pIV: pointer; IV_len: PtrInt; pAAD: pointer; aLen: PtrInt; ptp, ctp: Pointer; pLen: PtrInt; out tag: TAESBlock): boolean;

Single call AES-GCM encryption and authentication process
function Init(const Key; KeyBits: PtrInt): boolean;
    // Initialize the AES-GCM structure for the supplied Key

function Reset(pIV: pointer; IV_len: PtrInt): boolean;
    // Start AES-GCM encryption with a given Initialization Vector
    // IV_len is in bytes use 12 for exact IV setting, otherwise the supplied buffer will be hashed using gf_mul_h()

procedure Done;
    // Flush the AES context to avoid forensic issues
    // do nothing if Final() has been already called

TAESIVCTR = packed record
    // Used internally by TAESAbstract to detect replay attacks
    // when EncryptPKCS7/DecryptPKCS7 are used with IVAtBeginning=true, and IVReplayAttackCheck property contains repCheckedIfAvailable or repMandatory
    // EncryptPKCS7 will encrypt this record (using the global shared AESIVCTR_KEY over AES-128) to create a random IV, as a secure cryptographic pseudorandom number generator (CSPRNG), nonce and ctr ensuring 96 bits of entropy
    // DecryptPKCS7 will decode and ensure that the IV has an increasing CTR
    // memory size matches an TAESBlock on purpose, for direct encryption

    ctr: cardinal;
        // An increasing counter, used to detect replay attacks
        // is set to a 32-bit random value at initialization
        // is increased by one for every EncryptPKCS7, so can be checked against replay attack in DecryptPKCS7, and implement a safe CSPRNG for stored IV

    magic: cardinal;
        // Contains the crc32c hash of the block cipher mode (e.g. 'AESCFB')
        // when magic won't match (i.e. in case of mORMot revision < 3063), the check won't be applied in DecryptPKCS7: this security feature is backward compatible if IVReplayAttackCheck is repCheckedIfAvailable, but will fail for repMandatory

    nonce: QWord;
        // 8 bytes of random value

TAESAbstract = class(TSynPersistent)
    // Handle AES cypher/uncypher with chaining

    constructor Create(const aKey: THash256); reintroduce; overload;
        // Initialize AES context for AES-256 cypher
        // first method to call before using this class
        // just a wrapper around Create(aKey,256);
constructor Create(const aKey: THash128); reintroduce; overload;
  *Initialize AES context for AES-128 cypher*
  - first method to call before using this class
  - just a wrapper around Create(aKey, 128);

constructor Create(const aKey; aKeySize: cardinal); reintroduce; overload; virtual;
  *Initialize AES context for cypher*
  - first method to call before using this class
  - KeySize is in bits, i.e. 128, 192, 256

constructor CreateFromPBKDF2(const aKey: RawUTF8; const aSalt: RawByteString; aRounds: Integer);
  *Initialize AES context for cypher, from PBKDF2_HMAC_SHA256 derivation*
  - here the Key is supplied as a string, and will be hashed using PBKDF2_HMAC_SHA256 with the specified salt and rounds

constructor CreateFromSha256(const aKey: RawUTF8);
  *Initialize AES context for cypher, from SHA-256 hash*
  - here the Key is supplied as a string, and will be hashed using SHA-256 via the SHA256Weak proprietary algorithm - to be used only for backward compatibility of existing code
  - consider using more secure (and more standard) CreateFromPBKDF2 instead

constructor CreateTemp(aKeySize: cardinal);
  *Initialize AES context for cypher, from some TAESPRNG random bytes*
  - may be used to hide some sensitive information from memory, like CryptDataForCurrentUser but with a temporary key

destructor Destroy; override;
  *Release the used instance memory and resources*
  - also fill the secret fKey buffer with zeros, for safety

function AlgoName: TShort16;
  / returns e.g. 'aes128cfb' or '' if nil

function Clone: TAESAbstract; virtual;
  *Compute a class instance similar to this one*
  - could be used to have a thread-safe re-use of a given encryption key

function CloneEncryptDecrypt: TAESAbstract; virtual;
  *Compute a class instance similar to this one, for performing the reverse encryption/decryption process*
  - this default implementation calls Clone, but CFB/OFB/CTR chaining modes using only AES encryption (i.e. inheriting from TAESAbstractEncryptOnly) will return self to avoid creating two instances
  - warning: to be used only with IVAtBeginning=false
```pascal
function DecryptPKCS7(const Input: RawByteString; IVAtBeginning: boolean=false; RaiseESynCryptoOnError: boolean=true): RawByteString; overload;

Decrypt a memory buffer using a PKCS7 padding pattern
- PKCS7 padding is described in RFC 5652 - it will trim up to 16 bytes from the input buffer; note this method uses the padding only, not the whole PKCS#7 Cryptographic Message Syntax
- if IVAtBeginning is TRUE, the Initialization Vector will be taken from the beginning of the input binary buffer - if IVReplayAttackCheck is set, this IV will be validated to contain an increasing encrypted CTR, and raise an ESynCrypto when a replay attack attempt is detected
- if RaiseESynCryptoOnError=false, returns " on any decryption error

function DecryptPKCS7(const Input: TBytes; IVAtBeginning: boolean=false; RaiseESynCryptoOnError: boolean=true): TBytes; overload;

Decrypt a memory buffer using a PKCS7 padding pattern
- PKCS7 padding is described in RFC 5652 - it will trim up to 16 bytes from the input buffer; note this method uses the padding only, not the whole PKCS#7 Cryptographic Message Syntax
- if IVAtBeginning is TRUE, the Initialization Vector will be taken from the beginning of the input binary buffer - if IVReplayAttackCheck is set, this IV will be validated to contain an increasing encrypted CTR, and raise an ESynCrypto when a replay attack attempt is detected
- if RaiseESynCryptoOnError=false, returns [] on any decryption error

function DecryptPKCS7Buffer(Input: Pointer; InputLen: integer; IVAtBeginning: boolean; RaiseESynCryptoOnError: boolean=true): RawByteString;

Decrypt a memory buffer using a PKCS7 padding pattern
- PKCS7 padding is described in RFC 5652 - it will trim up to 16 bytes from the input buffer; note this method uses the padding only, not the whole PKCS#7 Cryptographic Message Syntax
- if IVAtBeginning is TRUE, the Initialization Vector will be taken from the beginning of the input binary buffer - this IV will in fact contain an internal encrypted CTR, to detect any replay attack attempt
- if RaiseESynCryptoOnError=false, returns " on any decryption error

function EncryptPKCS7(const Input: TBytes; IVAtBeginning: boolean=false): TBytes; overload;

Encrypt a memory buffer using a PKCS7 padding pattern
- PKCS7 padding is described in RFC 5652 - it will add up to 16 bytes to the input buffer; note this method uses the padding only, not the whole PKCS#7 Cryptographic Message Syntax
- if IVAtBeginning is TRUE, a random Initialization Vector will be computed, and stored at the beginning of the output binary buffer - this IV may contain an internal encrypted CTR, to detect any replay attack attempt, if IVReplayAttackCheck is set to repCheckedIfAvailable or repMandatory

function EncryptPKCS7(const Input: RawByteString; IVAtBeginning: boolean=false): RawByteString; overload;

Encrypt a memory buffer using a PKCS7 padding pattern
- PKCS7 padding is described in RFC 5652 - it will add up to 16 bytes to the input buffer; note this method uses the padding only, not the whole PKCS#7 Cryptographic Message Syntax
- if IVAtBeginning is TRUE, a random Initialization Vector will be computed, and stored at the beginning of the output binary buffer - this IV may contain an internal encrypted CTR, to detect any replay attack attempt, if IVReplayAttackCheck is set to repCheckedIfAvailable or repMandatory
```
function EncryptPKCS7Buffer(Input, Output: Pointer; InputLen, OutputLen: cardinal; IVAtBeginning: boolean): boolean;

Encrypt a memory buffer using a PKCS7 padding pattern
- PKCS7 padding is described in RFC 5652 - it will add up to 16 bytes to the input buffer; note this method uses the padding only, not the whole PKCS#7 Cryptographic Message Syntax
- use EncryptPKCS7Length() function to compute the actual needed length
- if IVAtBeginning is TRUE, a random Initialization Vector will be computed, and stored at the beginning of the output binary buffer - this IV will in fact contain an internal encrypted CTR, to detect any replay attack attempt
- returns TRUE on success, FALSE if OutputLen is not correct - you should use EncryptPKCS7Length() to compute the exact needed number of bytes

function EncryptPKCS7Length(InputLen: cardinal; IVAtBeginning: boolean): cardinal;

Compute how many bytes would be needed in the output buffer, when encrypting using a PKCS7 padding pattern
- could be used to pre-compute the OutputLength for EncryptPKCS7Buffer()
- PKCS7 padding is described in RFC 5652 - it will add up to 16 bytes to the input buffer; note this method uses the padding only, not the whole PKCS#7 Cryptographic Message Syntax

function MACAndEncrypt(const Data: RawByteString; Encrypt: boolean): RawByteString;

Perform one step PKCS7 encryption/decryption and authentication with the current AES instance
- returns " on any (MAC) issue during decryption (Encrypt=false) or if this class does not support AEAD MAC
- as used e.g. by CryptDataForCurrentUser()
- do not use this abstract class method, but inherited TAESCFBCRC/TAESOFBCRC
- will store a header with its own CRC, so detection of most invalid formats (e.g. from fuzzing input) will occur before any AES/MAC process

class function MACEncrypt(const Data: RawByteString; const Key: THash256; Encrypt: boolean): RawByteString; overload;

Perform one step PKCS7 encryption/decryption and authentication from a given 256-bit key
- returns " on any (MAC) issue during decryption (Encrypt=false) or if this class does not support AEAD MAC
- as used e.g. by CryptDataForCurrentUser()
- do not use this abstract class method, but inherited TAESCFBCRC/TAESOFBCRC
- will store a header with its own CRC, so detection of most invalid formats (e.g. from fuzzing input) will occur before any AES/MAC process

class function MACEncrypt(const Data: RawByteString; const Key: THash128; Encrypt: boolean): RawByteString; overload;

Perform one step PKCS7 encryption/decryption and authentication from a given 128-bit key
- returns " on any (MAC) issue during decryption (Encrypt=false) or if this class does not support AEAD MAC
- do not use this abstract class method, but inherited TAESCFBCRC/TAESOFBCRC
- will store a header with its own CRC, so detection of most invalid formats (e.g. from fuzzing input) will occur before any AES/MAC process
function MACEquals(const aCRC: THash256): boolean; virtual;

Validate if the computed AEAD MAC matches the expected supplied value
- is just a wrapper around MACGetLast() and IsEqual() functions

function MACGetLast(out aCRC: THash256): boolean; virtual;

Returns AEAD (authenticated-encryption with associated-data) MAC
- i.e. optional 256-bit MAC computation during last Encrypt/Decrypt call
- may be used e.g. for AES-GCM or our custom AES-CTR modes
- default implementation, for a non AEAD protocol, returns false

function MACSetNonce(const aKey: THash256; aAssociated: pointer=nil; aAssociatedLen: integer=0): boolean; virtual;

Initialize AEAD (authenticated-encryption with associated-data) nonce
- i.e. setup 256-bit MAC computation during next Encrypt/Decrypt call
- may be used e.g. for AES-GCM or our custom AES-CTR modes
- default implementation, for a non AEAD protocol, returns false

class function SimpleEncrypt(const Input: RawByteString; const Key; KeySize: integer; Encrypt: boolean; IVAtBeginning: boolean=false; RaiseESynCryptoOnError: boolean=true): RawByteString; overload;

Simple wrapper able to cypher/decypher any in-memory content
- here data variables could be text or binary
- you could use e.g. THMAC_SHA256 to safely compute the Key/KeySize value
- if IVAtBeginning is TRUE, a random Initialization Vector will be computed, and stored at the beginning of the output binary buffer
- will use SHA256Weak() and PKCS7 padding with the current class mode

class function SimpleEncrypt(const Input,Key: RawByteString; Encrypt: boolean; IVAtBeginning: boolean=false; RaiseESynCryptoOnError: boolean=true): RawByteString; overload;

Simple wrapper able to cypher/decypher any in-memory content
- here data variables could be text or binary
- use StringToUTF8() to define the Key parameter from a VCL string
- if IVAtBeginning is TRUE, a random Initialization Vector will be computed, and stored at the beginning of the output binary buffer
- will use SHA256Weak() and PKCS7 padding with the current class mode

class function SimpleEncryptFile(const InputFile, Outputfile: TFileName; const Key; KeySize: integer; Encrypt: boolean; IVAtBeginning: boolean=false; RaiseESynCryptoOnError: boolean=true): boolean; overload;

Simple wrapper able to cypher/decypher any file content
- just a wrapper around SimpleEncrypt() and StringFromFile/FileFromString
- you could use e.g. THMAC_SHA256 to safely compute the Key/KeySize value
- if IVAtBeginning is TRUE, a random Initialization Vector will be computed, and stored at the beginning of the output binary buffer
- will use SHA256Weak() and PKCS7 padding with the current class mode
class function SimpleEncryptFile(const InputFile, OutputFile: TFileName; const Key: RawByteString; Encrypt: boolean; IVAtBeginning: boolean=false; RaiseESynCryptoOnError: boolean=true): boolean; overload;

Simple wrapper able to cypher/decypher any file content
- just a wrapper around SimpleEncrypt() and StringFromFile/FileFromString
- use StringToUTF8() to define the Key parameter from a VCL string
- if IVAtBeginning is TRUE, a random Initialization Vector will be computed, and stored at the beginning of the output binary buffer
- will use SHA256Weak() and PKCS7 padding with the current class mode

procedure Decrypt(BufIn, BufOut: pointer; Count: cardinal); virtual; abstract;
Perform the AES un-cypher in the corresponding mode
- when used in block chaining mode, you should have set the IV property

procedure Encrypt(BufIn, BufOut: pointer; Count: cardinal); virtual; abstract;
Perform the AES cypher in the corresponding mode
- when used in block chaining mode, you should have set the IV property

property IV: TAESBlock read fIV write fIV;
Associated Initialization Vector
- all modes (except ECB) do expect an IV to be supplied for chaining, before any encryption or decryption is performed
- you could also use PKCS7 encoding with IVAtBeginning=true option

property IVHistoryDepth: integer read fIVHistoryDec.Depth write SetIVHistory;
Maintains an history of previous IV, to avoid re-play attacks
- only useful when EncryptPKCS7/DecryptPKCS7 are used with IVAtBeginning=true, and IVReplayAttackCheck is left to repNoCheck

property IVReplayAttackCheck: TAESIVReplayAttackCheck read fIVReplayAttackCheck write fIVReplayAttackCheck;
Let IV detect replay attack for EncryptPKCS7 and DecryptPKCS7
- if IVAtBeginning=true and this property is set, EncryptPKCS7 will store a random IV from an internal CTR, and DecryptPKCS7 will check this incoming IV CTR consistency, and raise an ESynCrypto exception on failure
- leave it to its default repNoCheck if the very same TAESAbstract instance is expected to be used with several sources, by which the IV CTR will be unsynchronized
- security warning: by design, this is NOT cautious with CBC chaining: you should use it only with CFB, OFB or CTR mode, since the IV sequence will be predictable if you know the fixed AES private key of this unit, but the IV sequence features uniqueness as it is generated by a good PRNG - see http://crypto.stackexchange.com/q/3515

property KeySize: cardinal read fKeySize;
Associated Key Size, in bits (i.e. 128,192,256)
TAESAbstractSyn = class(TAESAbstract)

Handle AES cypher/uncypher with chaining with out own optimized code
- use any of the inherited implementation, corresponding to the chaining mode required -
  TAESECB, TAESCBC, TAESCFB, TAESOFB and TAESCTR classes to handle in ECB, CBC, CFB, OFB and
  CTR mode (including PKCS7-like padding)
- this class will use AES-NI hardware instructions, if available
- those classes are re-entrant, i.e. that you can call the Encrypt* or Decrypt* methods on the same
  instance several times

  destructor Destroy; override;
  Release the used instance memory and resources
  - also fill the TAES instance with zeros, for safety

  function Clone: TAESAbstract; override;
  Creates a new instance with the very same values
  - by design, our classes will use TAES stateless context, so this method will just copy the current
    fields to a new instance, by-passing the key creation step

  procedure Decrypt(BufIn, BufOut: pointer; Count: cardinal); override;
  Perform the AES un-cypher in the corresponding mode
  - this abstract method will set CV from fIV property, and fln/fOut from BufIn/BufOut

  procedure Encrypt(BufIn, BufOut: pointer; Count: cardinal); override;
  Perform the AES cypher in the corresponding mode, over Count bytes
  - this abstract method will set CV from fIV property, and fln/fOut from BufIn/BufOut

  property CV: TAESBlock read fCV;
  Read-only access to the internal CV block, which may be have just been used by Encrypt/Decrypt
  methods

TAESECB = class(TAESAbstractSyn)

Handle AES cypher/uncypher without chaining (ECB)
- this mode is known to be less secure than the others
- IV property should be set to a fixed value to encode the trailing bytes of the buffer by a simple
  XOR - but you should better use the PKC7 pattern
- this class will use AES-NI hardware instructions, if available, e.g.
  ECB128: 19.70ms in x86 optimized code, 6.97ms with AES-NI

procedure Decrypt(BufIn, BufOut: pointer; Count: cardinal); override;
  Perform the AES un-cypher in the ECB mode

procedure Encrypt(BufIn, BufOut: pointer; Count: cardinal); override;
  Perform the AES cypher in the ECB mode
TAESCBC = class(TAESAbstractSyn)

Handle AES cypher/uncypher with Cipher-block chaining (CBC)
- this class will use AES-NI hardware instructions, if available, e.g.
  CBC192: 24.91ms in x86 optimized code, 9.75ms with AES-NI
  - expect IV to be set before process, or IVAtBeginning=true

procedure Decrypt(BufIn, BufOut: pointer; Count: cardinal); override;
  Perform the AES un-cypher in the CBC mode

procedure Encrypt(BufIn, BufOut: pointer; Count: cardinal); override;
  Perform the AES cypher in the CBC mode

TAESAbstractEncryptOnly = class(TAESAbstractSyn)

Abstract parent class for chaining modes using only AES encryption

constructor Create(const aKey; aKeySize: cardinal); override;
  Initialize AES context for cypher
  - will pre-generate the encryption key (aKeySize in bits, i.e. 128,192,256)

function CloneEncryptDecrypt: TAESAbstract; override;
  Compute a class instance similar to this one, for performing the reverse encryption/decryption process
  - will return self to avoid creating two instances
  - warning: to be used only with IVAtBeginning=false

TAESCFB = class(TAESAbstractEncryptOnly)

Handle AES cypher/uncypher with Cipher feedback (CFB)
- this class will use AES-NI hardware instructions, if available, e.g.
  CFB128: 22.25ms in x86 optimized code, 9.29ms with AES-NI
  - expect IV to be set before process, or IVAtBeginning=true

procedure Decrypt(BufIn, BufOut: pointer; Count: cardinal); override;
  Perform the AES un-cypher in the CFB mode

procedure Encrypt(BufIn, BufOut: pointer; Count: cardinal); override;
  Perform the AES cypher in the CFB mode

TAESOFB = class(TAESAbstractEncryptOnly)

Handle AES cypher/uncypher with Output feedback (OFB)
- this class will use AES-NI hardware instructions, if available, e.g.
  OFB256: 27.69ms in x86 optimized code, 9.94ms with AES-NI
  - expect IV to be set before process, or IVAtBeginning=true
  - TAESOFB 128/256 have an optimized asm version under x86_64 + AES_NI

procedure Decrypt(BufIn, BufOut: pointer; Count: cardinal); override;
  Perform the AES un-cypher in the OFB mode
procedure Encrypt(BufIn, BufOut: pointer; Count: cardinal); override;
    // Perform the AES cipher in the OFB mode

TAESCTR = class(TAESAbstractEncryptOnly)

Handle AES cipher/uncipher with 64-bit Counter mode (CTR)
- the CTR will use a counter in bytes 7..0 by default - which is safe but not standard - call
    ComposeIV() to change e.g. to NIST behavior
- this class will use AES-NI hardware instructions, e.g.
    CTR256: 28.13ms in x86 optimized code, 10.63ms with AES-NI
- expect IV to be set before process, or IVAtBeginning=true

constructor Create(const aKey; aKeySize: cardinal); override;
    // Initialize AES context for cypher
    // will pre-generate the encryption key (aKeySize in bits, i.e. 128,192,256)

function ComposeIV(const Nonce, Counter: TByteDynArray; LSBCounter: boolean): boolean; overload;
    // Defines how the IV is set and updated in CTR mode
    // you can specify startup Nonce and Counter, and the Counter position
    // Nonce + Counter lengths should add to 16 - otherwise returns false

function ComposeIV(Nonce, Counter: PAESBlock; NonceLen, CounterLen: integer; LSBCounter: boolean): boolean; overload;
    // Defines how the IV is set and updated in CTR mode
    // default (if you don't call this method) uses a Counter in bytes 7..0
    // you can specify startup Nonce and Counter, and the Counter position
    // NonceLen + CounterLen should be 16 - otherwise it fails and returns false

procedure Decrypt(BufIn, BufOut: pointer; Count: cardinal); override;
    // Perform the AES un-cipher in the CTR mode

procedure Encrypt(BufIn, BufOut: pointer; Count: cardinal); override;
    // Perform the AES cipher in the CTR mode

TAESMAC256 = record

    Internal 256-bit structure used for TAESAbstractAEAD MAC storage

    encrypted: THash128;
        // The plain MAC of the encrypted content
        // encrypted text digital signature, to check for errors, with no compromission of the plain content

    plain: THash128;
        // The AES-encrypted MAC of the plain content
        // plain text digital signature, to perform message authentication and integrity
TAESAbstractAEAD = class(TAESAbstractEncryptOnly)

\[ \text{AEAD (authenticated-encryption with associated-data) abstract class} \]
- perform AES encryption and on-the-fly MAC computation, i.e., computes a proprietary 256-bit MAC during AES cyphering, as 128-bit CRC of the encrypted data and 128-bit CRC of the plain data, seeded from a Key
- the 128-bit CRC of the plain text is then encrypted using the current AES engine, so returned 256-bit MAC has cryptographic level, and ensure data integrity, authenticity, and check against transmission errors

\[ \text{destructor Destroy; override;} \]
- Release the used instance memory and resources
- also fill the internal internal MAC hashes with zeros, for safety

\[ \text{function MACCheckError(aEncrypted: pointer; Count: cardinal): boolean; override;} \]
- Validate if an encrypted buffer matches the stored MAC
- expects the 256-bit MAC, as returned by MACGetLast, to be stored after the encrypted data
- returns true if the 128-bit CRC of the encrypted text matches the supplied buffer, ignoring the 128-bit CRC of the plain data
- since it is easy to forge such 128-CRC, it will only indicate that no transmission error occurred, but won't be an integrity or authentication proof (which will need full Decrypt + MACGetLast)
- may use any MACSetNonce() aAssociated value

\[ \text{function MACGetLast(out aCRC: THash256): boolean; override;} \]
- Returns 256-bit MAC computed during last Encrypt/Decrypt call
- encrypt the internal fMAC property value using the current AES cypher on the plain content and returns true; only the plain content CRC-128 is AES encrypted, to avoid reverse attacks against the known encrypted data

\[ \text{function MACSetNonce(const aKey: THash256; aAssociated: pointer=nil; aAssociatedLen: integer=0): boolean; override;} \]
- Initialize 256-bit MAC computation for next Encrypt/Decrypt call
- initialize the internal fMACKey property, and returns true
- only the plain text crc is seeded from aKey - encrypted message crc will use -1 as fixed seed, to avoid aKey compromise
- should be set with a new MAC key value before each message, to avoid replay attacks (as called from TECDHEProtocol.SetKey)

TAESCFBCRC = class(TAESAbstractAEAD)

\[ \text{AEAD combination of AES with Cipher feedback (CFB) and 256-bit MAC} \]
- this class will use AES-NI and CRC32C hardware instructions, if available
- expect IV to be set before process, or IVAtBeginning=true

\[ \text{procedure Decrypt(BufIn, BufOut: pointer; Count: cardinal); override;} \]
- Perform the AES un-cypher in the CFB mode, and compute 256-bit MAC

\[ \text{procedure Encrypt(BufIn, BufOut: pointer; Count: cardinal); override;} \]
- Perform the AES cypher in the CFB mode, and compute a 256-bit MAC
TAESOFBCRC = class(TAESAbstractAEAD)

*AEAD combination of AES with Output feedback (OFB) and 256-bit MAC*
- this class will use AES-NI and CRC32C hardware instructions, if available
- expect IV to be set before process, or IVAtBeginning=true

*procedure Decrypt(BufIn, BufOut: pointer; Count: cardinal); override;*
*Perform the AES un-cypher in the OFB mode, and compute a 256-bit MAC*

*procedure Encrypt(BufIn, BufOut: pointer; Count: cardinal); override;*
*Perform the AES cypher in the OFB mode, and compute a 256-bit MAC*

TAESGCM = class(TAESAbstract)

*Handle AES-GCM cypher/uncypher with built-in authentication*
- implements AEAD (authenticated-encryption with associated-data) methods like MACEncrypt/MACCheckError
- this class will use AES-NI hardware instructions, if available

*constructor Create(const aKey; aKeySize: cardinal); override;*
*Used to call AES.Reset() Initialize the AES-GCM context for cypher*
- first method to call before using this class
- KeySize is in bits, i.e. 128,192,256

*destructor Destroy; override;*
*Release the used instance memory and resources*
- also fill the internal TAES instance with zeros, for safety

*function Clone: TAESAbstract; override;*
*Creates a new instance with the very same values*
- by design, our classes will use TAESGCMEngine stateless context, so this method will just copy the current fields to a new instance, by-passing the key creation step

*function MACCheckError(aEncrypted: pointer; Count: cardinal): boolean; override;*
*Validate if an encrypted buffer matches the stored AEAD MAC*
- since AES-GCM is a one pass process, always assume the content is fine and returns true - we don't know the IV at this time

*function MACGetLast(out aCRC: THash256): boolean; override;*
*Returns AEAD (authenticated-encryption with associated-data) MAC*
- only the lower 128-bit (THash256.Lo) of aCRC is filled with the GMAC

*function MACSetNonce(const aKey: THash256; aAssociated: pointer=nil; aAssociatedLen: integer=0): boolean; override;*
*Prepare the AES-GCM process before Encrypt/Decrypt is called*
- aKey is not used: AES-GCM has its own nonce setting algorithm, and the IV will be set from random value by EncryptPKCS7()
- will just include any supplied associated data to the GMAC tag

*procedure Decrypt(BufIn, BufOut: pointer; Count: cardinal); override;*
*Perform the AES un-cypher and authentication*
procedure Encrypt(BufIn, BufOut: pointer; Count: cardinal); override;
Perform the AES-GCM cypher and authentication

TAESAbstract_API = class(TAESAbstract)
Handle AES cypher/uncypher using Windows CryptoAPI and the official Microsoft AES Cryptographic Provider (PROV_RSA_AES)
- see @http://msdn.microsoft.com/en-us/library/windows/desktop/aa386979
- timing of our optimized asm versions, for small (<=8KB) block processing (similar to standard web pages or most typical JSON/XML content), benchmarked on a Core i7 notebook and compiled as Win32 platform:

<table>
<thead>
<tr>
<th>Version</th>
<th>ECB</th>
<th>CBC</th>
<th>CFB</th>
<th>OFB</th>
<th>CTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES128</td>
<td>79.33ms</td>
<td>80.75ms</td>
<td>78.98ms</td>
<td>80.45ms</td>
<td></td>
</tr>
<tr>
<td>AES192</td>
<td>91.16ms</td>
<td>96.06ms</td>
<td>96.45ms</td>
<td>92.12ms</td>
<td>93.38ms</td>
</tr>
<tr>
<td>AES256</td>
<td>103.22ms</td>
<td>119.14ms</td>
<td>111.59ms</td>
<td>107.00ms</td>
<td>110.13ms</td>
</tr>
</tbody>
</table>

- timing of the same process, using CryptoAPI official PROV_RSA_AES provider:

<table>
<thead>
<tr>
<th>Version</th>
<th>ECB</th>
<th>CBC</th>
<th>CFB</th>
<th>OFB</th>
<th>CTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES128</td>
<td>102.88ms</td>
<td>124.91ms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES192</td>
<td>115.75ms</td>
<td>129.95ms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES256</td>
<td>139.50ms</td>
<td>154.02ms</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- but the CryptoAPI does not supports AES-NI, whereas our classes handle it, with a huge speed benefit
- under Win64, the official CryptoAPI is faster than our PUREPASCAL version, and the Win32 version of CryptoAPI itself, but slower than our AES-NI code

<table>
<thead>
<tr>
<th>Version</th>
<th>ECB</th>
<th>CBC</th>
<th>CFB</th>
<th>OFB</th>
<th>CTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES128</td>
<td>107.95ms</td>
<td>109.62ms</td>
<td>107.23ms</td>
<td>109.42ms</td>
<td></td>
</tr>
<tr>
<td>AES192</td>
<td>130.36ms</td>
<td>133.04ms</td>
<td>128.78ms</td>
<td>127.25ms</td>
<td>130.22ms</td>
</tr>
<tr>
<td>AES256</td>
<td>145.33ms</td>
<td>147.01ms</td>
<td>148.36ms</td>
<td>145.96ms</td>
<td>149.67ms</td>
</tr>
<tr>
<td>AES128</td>
<td>99.05ms</td>
<td>105.85ms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES256</td>
<td>107.11ms</td>
<td>118.04ms</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- in practice, you could forget about using the CryptoAPI, unless you are required to do so, for legal/corporate reasons

constructor Create(const aKey; aKeySize: cardinal); override;
Initialize AES context for cypher
- first method to call before using this class
- KeySize is in bits, i.e. 128,192,256

destructor Destroy; override;
Release the AES execution context

procedure Decrypt(BufIn, BufOut: pointer; Count: cardinal); override;
Perform the AES un-cypher in the ECB mode
- if Count is not a multiple of a 16 bytes block, the IV will be used to XOR the trailing bytes - so it won't be compatible with our TAESAbstractSyn classes: you should better use PKC7 padding instead

procedure Encrypt(BufIn, BufOut: pointer; Count: cardinal); override;
Perform the AES cypher in the ECB mode
- if Count is not a multiple of a 16 bytes block, the IV will be used to XOR the trailing bytes - so it won't be compatible with our TAESAbstractSyn classes: you should better use PKC7 padding instead

TAES ECB_API = class(TAESAbstract_API)
Handle AES cypher/uncypher without chaining (ECB) using Windows CryptoAPI
TAESCBC_API = class(TAESAbstract_API)
Handle AES cypher/uncypher Cipher-block chaining (CBC) using Windows CryptoAPI

TAESCFB_API = class(TAESAbstract_API)
Handle AES cypher/uncypher Cipher feedback (CFB) using Windows CryptoAPI
- NOT TO BE USED: the current PROV_RSA_AES provider does not return expected values for CFB

TAESOFB_API = class(TAESAbstract_API)
Handle AES cypher/uncypher Output feedback (OFB) using Windows CryptoAPI
- NOT TO BE USED: the current PROV_RSA_AES provider does not implement this mode, and returns a NTE_BAD_ALGID error

TAESLocked = class(TSynPersistentLock)
Thread-safe class containing a TAES encryption/decryption engine
destructor Destroy; override;
Finalize all used memory and resources

TAESPRNG = class(TAESLocked)
Cryptographic pseudorandom number generator (CSPRNG) based on AES-256
- use as a shared instance via TAESPRNG.Fill() overloaded class methods
- this class is able to generate some random output by encrypting successive values of a counter with AES-256 and a secret key
- this internal secret key is generated from PBKDF2 derivation of OS-supplied entropy using HMAC over SHA-512
- by design, such a PRNG is as good as the cypher used - for reference, see
https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator
- it would use fast hardware AES-NI or Padlock opcodes, if available

constructor Create(PBKDF2Rounds: integer = 16; ReseedAfterBytes: integer = 1024*1024; AESKeySize: integer = 256); reintroduce; virtual;
Initialize the internal secret key, using Operating System entropy
- entropy is gathered from the OS, using GetEntropy() method
- you can specify how many PBKDF2_HMAC_SHA512 rounds are applied to the OS-gathered entropy - the higher, the better, but also the slower
- internal private key would be re-seeded after ReseedAfterBytes bytes (1MB by default) are generated, using GetEntropy()
- by default, AES-256 will be used, unless AESKeySize is set to 128, which may be slightly faster (especially if AES-NI is not available)

function AFSplit(const Buffer: RawByteString; StripesCount: integer): RawByteString; overload;
Create an anti-forensic representation of a key for safe storage
- a binary buffer will be split into StripesCount items, ready to be saved on disk; returned length is BufferBytes*(StripesCount+1) bytes
- just a wrapper around the other overloaded AFSplit() function
function AFSplit(const Buffer; BufferBytes, StripesCount: integer): RawByteString; overload;

Create an anti-forensic representation of a key for safe storage
- a binary buffer will be split into StripesCount items, ready to be saved on disk; returned length is BufferBytes*(StripesCount+1) bytes
- AFSplit supports secure data destruction crucial for secure on-disk key management. The key idea is to bloat information and therefore improve the chance of destroying a single bit of it. The information is bloated in such a way, that a single missing bit causes the original information become unrecoverable.
- this implementation uses SHA-256 as diffusion element, and the current TAESPRNG instance to gather randomness
- for reference, see TKS1 as used for LUKS and defined in @https://gitlab.com/cryptsetup/cryptsetup/wikis/TKS1-draft.pdf

class function AFUnsplit(const Split: RawByteString; StripesCount: integer): RawByteString; overload;

Retrieve a key from its anti-forensic representation
- is the reverse function of AFSplit() method
- returns the un-splitted binary content
- returns " if StripesCount is incorrect

class function AFUnsplit(const Split: RawByteString; out Buffer; BufferBytes: integer): boolean; overload;

Retrieve a key from its anti-forensic representation
- is the reverse function of AFSplit() method
- returns TRUE if the input buffer matches BufferBytes value

class function Bytes(Len: integer): TBytes;

Just a wrapper around TAESPRNG.Main.FillRandomBytes() function
- this method is thread-safe, but you may use your own TAESPRNG instance if you need some custom entropy level

class function Fill(Len: integer): RawByteString; overload;

Just a wrapper around TAESPRNG.Main.FillRandom() function
- this method is thread-safe, but you may use your own TAESPRNG instance if you need some custom entropy level

function FillRandom(Len: integer): RawByteString; overload;

Returns a binary buffer filled with some pseudorandom data
- this method is thread-safe

function FillRandomBytes(Len: integer): TBytes;

Returns a binary buffer filled with some pseudorandom data
- this method is thread-safe

function FillRandomHex(Len: integer): RawUTF8;

Returns an hexa-encoded binary buffer filled with some pseudorandom data
- this method is thread-safe
class function GetEntropy(Len: integer; SystemOnly: boolean=false): RawByteString;

Retrieves some entropy bytes from the Operating System
- entropy comes from CryptGenRandom API on Windows, and /dev/urandom or /dev/random on Linux/POSIX
- this system-supplied entropy is then XORed with the output of a SHA-3 cryptographic SHAKE-256 generator in XOF mode, of several entropy sources (timestamp, thread and system information, SynCommons.Random32 function) unless SystemOnly is TRUE
- depending on the system, entropy may not be true randomness: if you need some truly random values, use TAESPRNG.Main.FillRandom() or TAESPRNG.Fill() methods, NOT this class function (which will be much slower, BTW)

class function Main: TAESPRNG;

Returns a shared instance of a TAESPRNG instance
- if you need to generate some random content, just call the TAESPRNG.Main.FillRandom() overloaded methods, or directly TAESPRNG.Fill()

function Random32(max: cardinal): cardinal; overload;

Returns a 32-bit unsigned random number, with a maximum value

function Random32: cardinal; overload;

Returns a 32-bit unsigned random number

function Random64: QWord;

Returns a 64-bit unsigned random number

function RandomDouble: double;

Returns a 64-bit floating-point random number in range [0..1]

function RandomExt: TSynExtended;

Returns a floating-point random number in range [0..1]

function RandomPassword(Len: integer): RawUTF8;

Computes a random ASCII password
- will contain uppercase/lower letters, digits and $.:()?%!+-*/@# excluding ;,= to allow direct use in CSV content

class procedure Fill(Buffer: pointer; Len: integer); overload;

Just a wrapper around TAESPRNG.Main.FillRandom() function
- this method is thread-safe, but you may use your own TAESPRNG instance if you need some custom entropy level

class procedure Fill(out Block: TAESBlock); overload;

Just a wrapper around TAESPRNG.Main.FillRandom() function
- this method is thread-safe, but you may use your own TAESPRNG instance if you need some custom entropy level

class procedure Fill(out Block: THash256); overload;

Just a wrapper around TAESPRNG.Main.FillRandom() function
- this method is thread-safe, but you may use your own TAESPRNG instance if you need some custom entropy level
procedure FillRandom(out Buffer: THash256); overload;
    Fill a 256-bit buffer with some pseudorandom data
    - this method is thread-safe

procedure FillRandom(Buffer: pointer; Len: integer); overload; virtual;
    Fill a binary buffer with some pseudorandom data
    - this method is thread-safe

procedure FillRandom(out Block: TAESBlock); overload; virtual;
    Fill a TAESBlock with some pseudorandom data
    - could be used e.g. to compute an AES Initialization Vector (IV)
    - this method is thread-safe

procedure Seed; virtual;
    Would force the internal generator to re-seed its private key
    - avoid potential attacks on backward or forward security
    - would be called by FillRandom() methods, according to SeedAfterBytes
    - this method is thread-safe

property AESKeySize: integer read fAESKeySize;
    How many bits (128 or 256 - which is the default) are used for the AES

property SeedAfterBytes: integer read fSeedAfterBytes;
    After how many generated bytes Seed method would be called
    - default is 1 MB

property SeedPBKDF2Rounds: cardinal read fSeedPBKDF2Rounds;
    How many PBKDF2_HMAC_SHA512 count is applied by Seed to the entropy
    - default is 16 rounds, which is more than enough for entropy gathering, since GetEntropy
    output comes from a SHAKE-256 generator in XOF mode

property TotalBytes: QWord read fTotalBytes;
    How many bytes this generator did compute

TAESPRNGSystem = class(TAESPRNG)
    TAESPRNG-compatible class using Operating System pseudorandom source
    - may be used instead of TAESPRNG if a "standard" generator is required - you could override
    MainAESPRNG global variable
    - will call /dev/urandom under POSIX, and CryptGenRandom API on Windows
    - warning: may block on some BSD flavors, depending on /dev/urandom
    - from the cryptographic point of view, our TAESPRNG class doesn't suffer from the "black-box"
    approach of Windows, give consistent randomness over all supported cross-platform, and is
    indubitably faster

constructor Create; reintroduce; virtual;
    Initialize the Operating System PRNG

procedure FillRandom(Buffer: pointer; Len: integer); override;
    Fill a binary buffer with some pseudorandom data
    - this method is thread-safe
procedure FillRandom(out Block: TAESBlock); override;

Fill a TAESBlock with some pseudorandom data
- this method is thread-safe

procedure Seed; override;

Called to force the internal generator to re-seed its private key
- won't do anything for the Operating System pseudorandom source

TSHA1 = object(TObject)
Handle SHA-1 hashing
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one initialized instance, e.g. for THMAC_SHA1
- see TSynHasher if you expect to support more than one algorithm at runtime

function Final(NoInit: boolean=false): TSHA1Digest; overload;

Finalize and compute the resulting SHA-1 hash Digest of all data affected to Update() method
- will also call Init to reset all internal temporary context, for safety

procedure Final(out Digest: TSHA1Digest; NoInit: boolean=false); overload;

Finalize and compute the resulting SHA-1 hash Digest of all data affected to Update() method
- will also call Init to reset all internal temporary context, for safety

procedure Full(Buffer: pointer; Len: integer; out Digest: TSHA1Digest);

One method to rule them all
- call Init, then Update(), then Final()
- only Full() is Padlock-implemented - use this rather than Update()

procedure Init;

Initialize SHA-1 context for hashing

procedure Update(Buffer: pointer; Len: integer); overload;

Update the SHA-1 context with some data

procedure Update(const Buffer: RawByteString); overload;

Update the SHA-1 context with some data

TSHA256 = object(TObject)
Handle SHA-256 hashing
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one initialized instance, e.g. for THMAC_SHA256
- see TSynHasher if you expect to support more than one algorithm at runtime

function Final(NoInit: boolean=false): TSHA256Digest; overload;

Finalize and compute the resulting SHA-256 hash Digest of all data affected to Update() method

procedure Final(out Digest: TSHA256Digest; NoInit: boolean=false); overload;

Finalize and compute the resulting SHA-256 hash Digest of all data affected to Update() method
procedure Full(Buffer: pointer; Len: integer; out Digest: TSHA256Digest);

One method to rule them all
- call Init, then Update(), then Final()
- only Full() is Padlock-implemented - use this rather than Update()

procedure Init;

Initialize SHA-256 context for hashing

procedure Update(Buffer: pointer; Len: integer); overload;

Update the SHA-256 context with some data

procedure Update(const Buffer: RawByteString); overload;

Update the SHA-256 context with some data

TSHA384 = object(TObject)

Handle SHA-384 hashing
- it is in fact a TSHA512 truncated hash, with other initial hash values
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one initialized instance, e.g. for THMAC_SHA384
- see TSynHasher if you expect to support more than one algorithm at runtime

function Final(NoInit: boolean=false): TSHA384Digest; overload;

Finalize and compute the resulting SHA-384 hash Digest of all data affected to Update() method

procedure Final(out Digest: TSHA384Digest; NoInit: boolean=false); overload;

Finalize and compute the resulting SHA-384 hash Digest of all data affected to Update() method
- will also call Init to reset all internal temporary context, for safety

procedure Full(Buffer: pointer; Len: integer; out Digest: TSHA384Digest);

One method to rule them all
- call Init, then Update(), then Final()

procedure Init;

Initialize SHA-384 context for hashing

procedure Update(Buffer: pointer; Len: integer); overload;

Update the SHA-384 context with some data

procedure Update(const Buffer: RawByteString); overload;

Update the SHA-384 context with some data
TSHA512 = object(TObject)

Handle SHA-512 hashing
- by design, this algorithm is expected to be much faster on 64-bit CPU, since all internal process involves QWord - but we included a SSE3 asm optimized version on 32-bit CPU under Windows and Linux, which is almost as fast as on plain x64, and even faster than SHA-256 and SHA-3
- under x86/Delphi, plain pascal is 40MB/s, SSE3 asm 180MB/s
- on x64, pascal Delphi is 150MB/s, and FPC is 190MB/s (thanks to native RorQWord intrinsic compiler function) - we also included a SSE4 asm version which outperforms other cryptographic hashes to more than 380MB/s
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one initialized instance, e.g. for THMAC_SHA512
- see TSynHasher if you expect to support more than one algorithm at runtime

function Final(NoInit: boolean=false): TSHA512Digest; overload;
  Finalize and compute the resulting SHA-512 hash Digest of all data affected to Update() method

procedure Final( out Digest: TSHA512Digest; NoInit: boolean=false); overload;
  Finalize and compute the resulting SHA-512 hash Digest of all data affected to Update() method
  - will also call Init to reset all internal temporary context, for safety

procedure Full(Buffer: pointer; Len: integer; out Digest: TSHA512Digest);
  One method to rule them all
  - call Init, then Update(), then Final()

procedure Init;
  Initialize SHA-512 context for hashing

procedure Update(Buffer: pointer; Len: integer); overload;
  Update the SHA-512 context with some data

procedure Update(const Buffer: RawByteString); overload;
  Update the SHA-512 context with some data

TSHA3 = object(TObject)

Handle SHA-3 (Keccak) hashing
- Keccak was the winner of the NIST hashing competition for a new hashing algorithm to provide an alternative to SHA-256. It became SHA-3 and was named by NIST a FIPS 180-4, then FIPS 202 hashing standard in 2015
- by design, SHA-3 doesn't need to be encapsulated into a HMAC algorithm, since it already includes proper padding, so keys could be concatenated
- this implementation is based on Wolfgang Ehrhardt's and Eric Grange's, with our own manually optimized x64 assembly
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one initialized instance, e.g. after InitCypher
- see TSynHasher if you expect to support more than one algorithm at runtime

function Algorithm: TSHA3Algo;
  Returns the algorithm specified at Init()
function Cypher(const Source: RawByteString): RawByteString; overload;

Uses SHA-3 in "Extendable-Output Function" (XOF) to cipher some content
- this overloaded function expects the instance to have been prepared by previous InitCypher call
- resulting string will have the very same size than the Source
- XOF is implemented as a symmetrical algorithm: use this Cypher() method for both encryption and decryption of any buffer
- you can call this method several times, to work with a stream buffer; but for safety, you should eventually call Done

function Cypher(const Key, Source: RawByteString; Algo: TSHA3Algo = SHAKE_256): RawByteString; overload;

Uses SHA-3 in "Extendable-Output Function" (XOF) to cipher some content
- this overloaded function works with RawByteString content
- resulting string will have the very same size than the Source
- XOF is implemented as a symmetrical algorithm: use this Cypher() method for both encryption and decryption of any buffer

function Final256(NoInit: boolean=false): THash256;
Finalize and compute the resulting SHA-3 hash 256-bit Digest

function Final512(NoInit: boolean=false): THash512;
Finalize and compute the resulting SHA-3 hash 512-bit Digest

function FullStr(Algo: TSHA3Algo; Buffer: pointer; Len: integer; DigestBits: integer=0): RawUTF8;
Compute a SHA-3 hash hexadecimal Digest from a buffer, in one call
- call Init, then Update(), then Final() using the supplied algorithm
- default DigestBits=0 will write the default number of bits to Digest output memory buffer, according to the specified TSHA3Algo

procedure Cypher(Source, Dest: pointer; DataLen: integer); overload;

Uses SHA-3 in "Extendable-Output Function" (XOF) to cipher some content
- this overloaded function expects the instance to have been prepared by previous InitCypher call
- resulting Dest buffer will have the very same size than the Source
- XOF is implemented as a symmetrical algorithm: use this Cypher() method for both encryption and decryption of any buffer
- you can call this method several times, to work with a stream buffer; but for safety, you should eventually call Done

procedure Cypher(Key, Source, Dest: pointer; KeyLen, DataLen: integer; Algo: TSHA3Algo = SHAKE_256); overload;

Uses SHA-3 in "Extendable-Output Function" (XOF) to cipher some content
- there is no MAC stored in the resulting binary
- Source and Dest will have the very same DataLen size in bytes, and Dest will be Source XORed with the XOF output, so encryption and decryption are just obtained by the same symmetric call
- in this implementation, Source and Dest should point to two diverse buffers
- for safety, the Key should be a secret value, pre-pended with a random salt/IV or a resource-specific identifier (e.g. a record ID or a S/N), to avoid reverse composition of the cypher from known content - note that concatenating keys with SHA-3 is as safe as computing a HMAC for SHA-2
procedure Done;

Fill all used memory context with zeros, for safety
- is necessary only when NoInit is set to true (e.g. after InitCypher)

procedure Final(out Digest: THash512; NoInit: boolean=false); overload;
Finalize and compute the resulting SHA-3 hash 512-bit Digest

procedure Final(out Digest: THash256; NoInit: boolean=false); overload;
Finalize and compute the resulting SHA-3 hash 256-bit Digest

procedure Final(Digest: pointer; DigestBits: integer=0; NoInit: boolean=false); overload;
Finalize and compute the resulting SHA-3 hash Digest
- Digest destination buffer must contain enough bytes
- default DigestBits=0 will write the default number of bits to Digest output memory buffer, according to the current TSHA3Algo
- you can call this method several times, to use this SHA-3 hasher as "Extendable-Output Function" (XOF), e.g. for stream encryption (ensure NoInit is set to true, to enable recall)

procedure Full(Buffer: pointer; Len: integer; out Digest: THash256); overload;
Compute a SHA-3 hash 256-bit Digest from a buffer, in one call
- call Init, then Update(), then Final() using SHA3_256 into a THash256

procedure Full(Buffer: pointer; Len: integer; out Digest: THash512); overload;
Compute a SHA-3 hash 512-bit Digest from a buffer, in one call
- call Init, then Update(), then Final() using SHA3_512 into a THash512

procedure Full(Algo: TSHA3Algo; Buffer: pointer; Len: integer; Digest: pointer; DigestBits: integer=0); overload;
Compute a SHA-3 hash Digest from a buffer, in one call
- call Init, then Update(), then Final() using the supplied algorithm
- default DigestBits=0 will write the default number of bits to Digest output memory buffer, according to the specified TSHA3Algo

procedure Init(Algo: TSHA3Algo);
Initialize SHA-3 context for hashing
- in practice, you may use SHA3_256 or SHA3_512 to return THash256 or THash512 digests

procedure InitCypher(Key: pointer; KeyLen: integer; Algo: TSHA3Algo = SHAKE_256); overload;
Uses SHA-3 in "Extendable-Output Function" (XOF) to cypher some content
- prepare the instance to further Cypher() calls
- you may reuse the very same TSHA3 instance by copying it to a local variable before calling this method (this copy is thread-safe)
- works with RawByteString content

procedure InitCypher(const Key: RawByteString; Algo: TSHA3Algo = SHAKE_256); overload;
Uses SHA-3 in "Extendable-Output Function" (XOF) to cypher some content
- prepare the instance to further Cypher() calls
- you may reuse the very same TSHA3 instance by copying it to a local variable before calling this method (this copy is thread-safe)
- works with RawByteString content
procedure Update(Buffer: pointer; Len: integer); overload;
Update the SHA-3 context with some data

procedure Update(const Buffer: RawByteString); overload;
Update the SHA-3 context with some data

TMD5 = object(TObject)
Handle MD5 hashing
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one
initialized instance
- see TSynHasher if you expect to support more than one algorithm at runtime
- even if MD5 is now seldom used, it is still faster than SHA alternatives, when you need a 128-bit
cryptographic hash, but can afford some collisions
- this implementation has optimized x86 and x64 assembly, for processing around 500MB/s, and a
pure-pascal fallback code on other platforms

function Final: TMD5Digest; overload;
Finalize and compute the resulting MD5 hash Digest of all data affected to Update() method

procedure Final(out result: TMD5Digest); overload;
Finalize and compute the resulting MD5 hash Digest of all data affected to Update() method

procedure Finalize;
Finalize the MD5 hash process
- the resulting hash digest would be stored in buf public variable

procedure Full(Buffer: pointer; Len: integer; out Digest: TMD5Digest);
One method to rule them all
- call Init, then Update(), then Final()

procedure Init;
Initialize MD5 context for hashing

procedure Update(const Buffer: RawByteString); overload;
Update the MD5 context with some data

procedure Update(const buffer; Len: cardinal); overload;
Update the MD5 context with some data

TRC4 = object(TObject)
Handle RC4 encryption/decryption
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one
initialized instance
- you can also restore and backup any previous state of the RC4 encryption by copying the whole
TRC4 variable into another (stack-allocated) variable

procedure Drop(Count: cardinal);
Drop the next Count bytes from the RC4 cypher state
- may be used in Stream mode, or to initialize in RC4-drop[n] mode
procedure Encrypt(const BufIn; var BufOut; Count: cardinal);
  Perform the RC4 cipher encryption/decryption on a buffer
  - each call to this method shall be preceded with an Init() call
  - RC4 is a symmetrical algorithm: use this Encrypt() method for both encryption and decryption
    of any buffer

procedure EncryptBuffer(BufIn, BufOut: PByte; Count: cardinal);
  Perform the RC4 cipher encryption/decryption on a buffer
  - each call to this method shall be preceded with an Init() call
  - RC4 is a symmetrical algorithm: use this EncryptBuffer() method for both encryption and decryption
    of any buffer

procedure Init(const aKey; aKeyLen: integer);
  Initialize the RC4 encryption/decryption
  - KeyLen is in bytes, and should be within 1..255 range

procedure InitSHA3(const aKey; aKeyLen: integer);
  Initialize RC4-drop[3072] encryption/decryption after SHA-3 hashing
  - will use SHAKE-128 generator in XOF mode to generate a 256 bytes key, then drop the first
    3072 bytes from the RC4 stream
  - this initializer is much safer than plain Init, so should be considered for any use on RC4 for new
    projects - even if AES-NI is 2 times faster, and safer SHAKE-128 operates in XOF mode at a similar
    speed range

TAESFullHeader = object(TObject)
  Internal header for storing our AES data with salt and CRC
  - memory size matches an TAESBlock on purpose, for direct encryption

  HeaderCheck: cardinal;
    CRC from header

  OriginalLen: cardinal;
    Len before compression (if any)

  SomeSalt: cardinal;
    Random Salt for better encryption

  SourceLen: cardinal;
    Len before AES encoding

function Calc(const Key; KeySize: cardinal): cardinal;
  Computes the Key checksum, using Adler32 algorithm

TAESFull = object(TObject)
  AES and XOR encryption object for easy direct memory or stream access
  - calls internally TAES obj methods, and handle memory and streams for best speed
  - a TAESFullHeader is encrypted at the begining, allowing fast Key validation, but the resulting
    stream is not compatible with raw TAES object
Head: TAESFullHeader;

  Header, stored at the beginning of struct -> 16-byte aligned

outStreamCreated: TMemoryStream;

  This memory stream is used in case of EncodeDecode(outStream=bOut=nil) method call

function EncodeDecode(const Key; KeySize, inLen: cardinal; Encrypt: boolean;
inStream, outStream: TStream; bIn, bOut: pointer; OriginalLen: cardinal=0): integer;

  Main method of AES or XOR cypher/uncypher
  - return out size, -1 if error on decoding (Key not correct)
  - valid KeySize: 0=nothing, 32=xor, 128,192,256=AES
  - if outStream is TMemoryStream -> auto-reserve space (no Realloc:)
  - for normal usage, you just have to Assign one In and one Out
  - if outStream AND bOut are both nil, an outStream is created via THeapMemoryStream.Create
  - if Padlock is used, 16-byte alignment is forced (via tmp buffer if necessary)
  - if Encrypt -> OriginalLen can be used to store unCompressed Len

TAESWriteStream = class(TStream)

  AES encryption stream
  - encrypt the Data on the fly, in a compatible way with AES() - last bytes are coded with XOR (not compatible with TAESFull format)
  - not optimized for small blocks -> ok if used AFTER TBZCompressor/TZipCompressor
  - warning: Write() will crypt Buffer memory in place -> use AFTER T*Compressor

  DestSize: cardinal;

    CRC from uncrypted compressed data - for Key check

constructor Create(outStream: TStream; const Key; KeySize: cardinal);

  If KeySize=0 initialize the AES encryption stream for an output stream (e.g. a TMemoryStream or a TFileStream)

destructor Destroy; override;

  Finalize the AES encryption stream
  - internally call the Finish method

function Read(var Buffer; Count: Longint): Longint; override;

  Read some data is not allowed -> this method will raise an exception on call

function Seek(Offset: Longint; Origin: Word): Longint; override;

  Read some data is not allowed -> this method will raise an exception on call

function Write(const Buffer; Count: Longint): Longint; override;

  Append some data to the outStream, after encryption

procedure Finish;

  Write pending data
  - should always be called before closeing the outStream (some data may still be in the internal buffers)
THMAC_SHA1 = object(TObject)

*Compute the HMAC message authentication code using SHA-1 as hash function*
- you may use HMAC_SHA1() overloaded functions for one-step process
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one initialized instance via Compute(), e.g. for fast PBKDF2

```pascal
procedure Compute(msg: pointer; msglen: integer; out result: TSHA1Digest);

*Computes the HMAC of the supplied message according to the key*
- expects a previous call on Init() to setup the shared key
- similar to a single Update(msg,msglen) followed by Done, but re-usable
- this method is thread-safe on any shared THMAC_SHA1 instance
```

```pascal
procedure Done(out result: RawUTF8; NoInit: boolean=false); overload;

*Computes the HMAC of all supplied message according to the key*
```

```pascal
procedure Done(out result: TSHA1Digest; NoInit: boolean=false); overload;

*Computes the HMAC of all supplied message according to the key*
```

```pascal
procedure Init(key: pointer; keylen: integer);

*Prepare the HMAC authentication with the supplied key*
- content of this record is stateless, so you can prepare a HMAC for a key using Init, then copy this THMAC_SHA1 instance to a local variable, and use this local thread-safe copy for actual HMAC computing
```

```pascal
procedure Update(msg: pointer; msglen: integer);

*Call this method for each continuous message block*
- iterate over all message blocks, then call Done to retrieve the HMAC
```

THMAC_SHA384 = object(TObject)

*Compute the HMAC message authentication code using SHA-384 as hash function*
- you may use HMAC_SHA384() overloaded functions for one-step process
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one initialized instance via Compute(), e.g. for fast PBKDF2

```pascal
procedure Compute(msg: pointer; msglen: integer; out result: TSHA384Digest);

*Computes the HMAC of the supplied message according to the key*
- expects a previous call on Init() to setup the shared key
- similar to a single Update(msg,msglen) followed by Done, but re-usable
- this method is thread-safe on any shared THMAC_SHA384 instance
```

```pascal
procedure Done(out result: RawUTF8; NoInit: boolean=false); overload;

*Computes the HMAC of all supplied message according to the key*
```

```pascal
procedure Done(out result: TSHA384Digest; NoInit: boolean=false); overload;

*Computes the HMAC of all supplied message according to the key*
**Synopse mORMot Framework**

*Software Architecture Design 1.18*

*Date: September 16, 2020*

---

**procedure Init(key: pointer; keylen: integer);**

*Prepare the HMAC authentication with the supplied key*
- content of this record is stateless, so you can prepare a HMAC for a key using Init, then copy this THMAC_SHA384 instance to a local variable, and use this local thread-safe copy for actual HMAC computing

**procedure Update(msg: pointer; msglen: integer);**

*Call this method for each continuous message block*
- iterate over all message blocks, then call Done to retrieve the HMAC

---

**THMAC_SHA512 = object(TObject)**

*Compute the HMAC message authentication code using SHA-512 as hash function*
- you may use HMAC_SHA512() overloaded functions for one-step process
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one initialized instance via Compute(), e.g. for fast PBKDF2

**procedure Compute(msg: pointer; msglen: integer; out result: TSHA512Digest);**

*Computes the HMAC of the supplied message according to the key*
- expects a previous call on Init() to setup the shared key
- similar to a single Update(msg, msglen) followed by Done, but re-usable
- this method is thread-safe on any shared THMAC_SHA512 instance

**procedure Done(out result: RawUTF8; NoInit: boolean=false); overload;**

*Computes the HMAC of all supplied message according to the key*

---

**THMAC_SHA256 = object(TObject)**

*Compute the HMAC message authentication code using SHA-256 as hash function*
- you may use HMAC_SHA256() overloaded functions for one-step process
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one initialized instance via Compute(), e.g. for fast PBKDF2

**procedure Compute(msg: pointer; msglen: integer; out result: TSHA256Digest);**

*Computes the HMAC of the supplied message according to the key*
- expects a previous call on Init() to setup the shared key
- similar to a single Update(msg, msglen) followed by Done, but re-usable
- this method is thread-safe on any shared THMAC_SHA256 instance


### procedure Done(out result: TSHA256Digest; NoInit: boolean=false); overload;
Computes the HMAC of all supplied message according to the key

### procedure Done(out result: RawUTF8; NoInit: boolean=false); overload;
Computes the HMAC of all supplied message according to the key

### procedure Init(key: pointer; keylen: integer);
Prepare the HMAC authentication with the supplied key
- Content of this record is stateless, so you can prepare a HMAC for a key using Init, then copy this THMAC_SHA256 instance to a local variable, and use this local thread-safe copy for actual HMAC computing

### procedure Update(const msg: RawByteString); overload;
Call this method for each continuous message block
- Iterate over all message blocks, then call Done to retrieve the HMAC

### procedure Update(msg: pointer; msglen: integer); overload;
Call this method for each continuous message block
- Iterate over all message blocks, then call Done to retrieve the HMAC

### procedure Update(const msg: THash128); overload;
Call this method for each continuous message block
- Iterate over all message blocks, then call Done to retrieve the HMAC

### procedure Update(const msg: THash256); overload;
Call this method for each continuous message block
- Iterate over all message blocks, then call Done to retrieve the HMAC

```pascal
TSynSignerParams = packed record
  JSON-serialization ready object as used by TSynSigner.PBKDF2 overloaded methods
  - Default value for unspecified parameters will be SHAKE_128 with rounds=1000 and a fixed salt

TSynSigner = object(TObject)
  A generic wrapper object to handle digital HMAC-SHA-2/SHA-3 signatures
  - Used e.g. to implement TJWTSynSignerAbstract

function Final: RawUTF8; overload;
  Returns the computed digital signature as lowercase hexadecimal text

function Full(aAlgo: TSignAlgo; const aSecret: RawUTF8; aBuffer: Pointer; aLen: integer): RawUTF8; overload;
  One-step digital signature of a buffer as lowercase hexadecimal string

function Full(aAlgo: TSignAlgo; const aSecret, aSalt: RawUTF8; aSecretPBKDF2Rounds: integer; aBuffer: Pointer; aLen: integer): RawUTF8; overload;
  One-step digital signature of a buffer with PBKDF2 derivation

procedure AssignTo(var aDerivatedKey: THash512Rec; out aAES: TAES; aEncrypt: boolean);
  Prepare a TAES object with the key derived via a PBKDF2() call
  - aDerivatedKey is defined as "var", since it will be zeroed after use
```
procedure Done;
  Fill the internal context with zeros, for security

procedure Final(out aSignature: THash512Rec; aNoInit: boolean=false); overload;
  Returns the raw computed digital signature
  - SignatureSize bytes will be written: use Signature.Lo/h0/b3/b accessors

procedure Init(aAlgo: TSignAlgo; const aSecret: RawUTF8); overload;
  Initialize the digital HMAC/SHA-3 signing context with some secret text

procedure Init(aAlgo: TSignAlgo; aSecret: pointer; aSecretLen: integer); overload;
  Initialize the digital HMAC/SHA-3 signing context with some secret binary

procedure Init(aAlgo: TSignAlgo; const aSecret, aSalt: RawUTF8;
aSecretPBKDF2Rounds: integer; aPBKDF2Secret: PHash512Rec=nil); overload;
  Initialize the digital HMAC/SHA-3 signing context with PBKDF2 safe iterative key derivation of a secret salted text

procedure PBKDF2(aParamsJSON: PUTF8Char; aParamsJSONLen: integer; out aDerivatedKey: THash512Rec; const aDefaultSalt: RawUTF8='I6sWioAidNnhXO9BK'; aDefaultAlgo: TSignAlgo=saSha3S128); overload;
  Convenient wrapper to perform PBKDF2 safe iterative key derivation
  - accept as input a TSynSignerParams serialized as JSON object

procedure PBKDF2(const aParamsJSON: RawUTF8; out aDerivatedKey: THash512Rec; const aDefaultSalt: RawUTF8='I6sWioAidNnhXO9BK'; aDefaultAlgo: TSignAlgo=saSha3S128); overload;
  Convenient wrapper to perform PBKDF2 safe iterative key derivation
  - accept as input a TSynSignerParams serialized as JSON object

procedure PBKDF2(aAlgo: TSignAlgo; const aSecret, aSalt: RawUTF8;
aSecretPBKDF2Rounds: integer; out aDerivatedKey: THash512Rec); overload;
  Convenient wrapper to perform PBKDF2 safe iterative key derivation

procedure PBKDF2(const aParams: TSynSignerParams; out aDerivatedKey: THash512Rec); overload;
  Convenient wrapper to perform PBKDF2 safe iterative key derivation

procedure Update(aBuffer: pointer; aLen: integer); overload;
  Process some message content supplied as memory buffer

procedure Update(const aBuffer: RawByteString); overload;
  Process some message content supplied as string

property Algo: TSignAlgo read fAlgo;
  The algorithm used for digital signature

property SignatureSize: integer read fSignatureSize;
  The size, in bytes, of the digital signature of this algorithm
  - potential values are 20, 28, 32, 48 and 64
TSynHasher = object(TObject)

Convenient multi-algorithm hashing wrapper
- as used e.g. by HashFile/HashFull functions
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one initialized instance

function Final: RawUTF8;
Returns the resulting hash as lowercase hexadecimal string

function Full(aAlgo: THashAlgo; aBuffer: Pointer; aLen: integer): RawUTF8;
One-step hash computation of a buffer as lowercase hexadecimal string

function Init(aAlgo: THashAlgo): boolean;
Enough space for all algorithms initialize the internal hashing structure for a specific algorithm
- returns false on unknown/unsupported algorithm

procedure Update(aBuffer: Pointer; aLen: integer); overload;
Hash the supplied memory buffer

procedure Update(const aBuffer: RawByteString); overload;
Hash the supplied string content

property Algo: THashAlgo read fAlgo;
The hash algorithm used by this instance

THMAC_CRC32C = object(TObject)

Compute the HMAC message authentication code using crc32c as hash function
- HMAC over a non cryptographic hash function like crc32c is known to be a safe enough MAC, if the supplied key comes e.g. from cryptographic HMAC_SHA256
- SSE 4.2 will let MAC be computed at 4 GB/s on a Core i7
- you may use HMAC_CRC32C() overloaded functions for one-step process
- we defined a record instead of a class, to allow stack allocation and thread-safe reuse of one initialized instance via Compute()

function Compute(msg: pointer; msglen: integer): cardinal;
Computes the HMAC of the supplied message according to the key
- expects a previous call on Init() to setup the shared key
- similar to a single Update(msg,msglen) followed by Done, but re-usable
- this method is thread-safe

function Done(NoInit: boolean=false): cardinal;
Computes the HMAC of all supplied message according to the key

procedure Init(key: pointer; keylen: integer); overload;
Prepare the HMAC authentication with the supplied key
- consider using Compute to re-use a prepared HMAC instance

procedure Init(const key: RawByteString); overload;
Prepare the HMAC authentication with the supplied key
- consider using Compute to re-use a prepared HMAC instance
procedure Update(msg: pointer; msglen: integer); overload;
Call this method for each continuous message block
- iterate over all message blocks, then call Done to retrieve the HMAC

procedure Update(const msg: RawByteString); overload;
Call this method for each continuous message block
- iterate over all message blocks, then call Done to retrieve the HMAC

IProtocol = interface(IInterface)
Perform safe communication after unilateral or mutual authentication
- see e.g. TProtocolNone or SynEcc's TECDEHProtocolClient and TECDEHProtocolServer implementation classes

function Clone: IProtocol;
Will create another instance of this communication protocol

function Decrypt(const aEncrypted: RawByteString; out aPlain: RawByteString): TProtocolResult;
Decrypt a message on one side, as transmitted from the other side
- should return sprSuccess if the
- should return sprInvalidMAC in case of wrong aEncrypted input (e.g. packet corruption, MiM or Replay attacks attempts)
- this method should be thread-safe in the implementation class

function ProcessHandshake(const MsgIn: RawUTF8; out MsgOut: RawUTF8): TProtocolResult;
Initialize the communication by exchanging some client/server information
- expects the handshaking messages to be supplied as UTF-8 text, may be as base64-encoded binary - see e.g. TWebSocketProtocolBinary.ProcessHandshake
- should return sprUnsupported if the implemented protocol does not expect any handshaking mechanism
- returns sprSuccess and set something into OutData, depending on the current step of the handshake
- returns an error code otherwise

procedure Encrypt(const aPlain: RawByteString; out aEncrypted: RawByteString);
Encrypt a message on one side, ready to be transmitted to the other side
- this method should be thread-safe in the implementation class

TProtocolNone = class(TInterfacedObject)
Implements a fake no-encryption protocol
- may be used for debugging purposes, or when encryption is not needed

function Clone: IProtocol;
Will create another instance of this communication protocol

function Decrypt(const aEncrypted: RawByteString; out aPlain: RawByteString): TProtocolResult;
Decrypt a message on one side, as transmitted from the other side
- this method will return the encrypted text with no actual decryption
function ProcessHandshake(const MsgIn: RawUTF8; out MsgOut: RawUTF8): TProtocolResult;
  \textit{Initialize the communication by exchanging some client/server information}
- this method will return \texttt{sprUnsupported}

procedure Encrypt(const aPlain: RawByteString; out aEncrypted: RawByteString);
  \textit{Encrypt a message on one side, ready to be transmitted to the other side}
- this method will return the plain text with no actual encryption

TProtocolAES = class(TInterfacedObjectLocked)
\textit{Implements a secure protocol using AES encryption}
- as used e.g. by 'synopsebinary' WebSockets protocol
- this class will maintain two TAESAbstract instances, one for encryption and another one for decryption, with PKCS7 padding and no MAC validation

constructor Create(aClass: TAESAbstractClass; const aKey; aKeySize: cardinal;
aIVReplayAttackCheck: TAESIVReplayAttackCheck=repCheckedIfAvailable);
  \texttt{reintroduce; virtual;}
  \texttt{[false]=decrypt [true]=encrypt initialize this encryption protocol with the given AES settings}

constructor CreateFrom(aAnother: TProtocolAES);
  \texttt{reintroduce; virtual;}
  Will create another instance of this communication protocol

destructor Destroy; override;
  Finalize the encryption

function Clone: IProtocol;
  Will create another instance of this communication protocol

function Decrypt(const aEncrypted: RawByteString; out aPlain: RawByteString): TProtocolResult;
  \textit{Decrypt a message on one side, as transmitted from the other side}
  - this method uses AES decryption and PKCS7 padding

function ProcessHandshake(const MsgIn: RawUTF8; out MsgOut: RawUTF8):
  \textit{TProtocolResult;}
  \textit{Initialize the communication by exchanging some client/server information}
  - this method will return \texttt{sprUnsupported}, since no key negociation is involved

procedure Encrypt(const aPlain: RawByteString; out aEncrypted: RawByteString);
  \textit{Encrypt a message on one side, ready to be transmitted to the other side}
  - this method uses AES encryption and PKCS7 padding

EJWTException = class(ESynException)
\textit{Exception raised when running JSON Web Tokens}

TJWTContent = record
\textit{JWT decoded content, as processed by TJWTAbstract}
- optionally cached in memory
  audience: \texttt{set of 0..15;}
  \texttt{Match TJWTAbstract.Audience[]} indexes for reg[jrcAudience]
claims: TJWTClaims;
Set of known/registered claims, as stored in the JWT payload

data: TDocVariantData;
Custom/unregistered claim values, as stored in the JWT payload
- registered claims will be available from reg[], not in this field
- e.g. data.U['name']='John Doe' and data.B['admin']=true for
  {"sub": "1234567890","name": "John Doe","admin": true} but data.U['sub'] if not defined, and reg[jrcSubject]='1234567890'

reg: array[TJWTClaim] of RawUTF8;
Known/registered claims UTF-8 values, as stored in the JWT payload
- e.g. reg[jrcSubject]='1234567890' and reg[jrcIssuer]='' for
  {"sub": "1234567890","name": "John Doe","admin": true}

result: TJWTResult;
Store latest Verify() result

TJWTAbstract = class(TSynPersistent)
Abstract parent class for implementing JSON Web Tokens
- to represent claims securely between two parties, as defined in industry standard
  @http://tools.ietf.org/html/rfc7519
- you should never use this abstract class directly, but e.g. TJWTHS256, TJWTHS384, TJWTHS512 or
  TJWTES256 (as defined in SynEcc.pas) inherited classes
- for security reasons, one inherited class is implementing a single algorithm, as is very likely to be
  the case on production: you pickup one "alg", then you stick to it; if your server needs more than
  one algorithm for compatibility reasons, use a separate key and URI - this design will reduce attack
  surface, and fully avoid weaknesses as described in
  @https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries and
  @http://tools.ietf.org/html/rfc7518#section-8.5

constructor Create(const aAlgorithm: RawUTF8; aClaims: TJWTClaims; const aAudience:
array of RawUTF8; aExpirationMinutes: integer; aIDIdentifier:
TSynUniqueIdentifierProcess; aIDObfuscationKey: RawUTF8); reintroduce;
Initiate the JWT processing instance
- the supplied set of claims are expected to be defined in the JWT payload
- aAudience are the allowed values for the jrcAudience claim
- aExpirationMinutes is the deprecation time for the jrcExpirationTime claim
- aIDIdentifier and aIDObfuscationKey are passed to a TSynUniqueIdentifierGenerator instance
  used for jrcJwtID claim

destructor Destroy; override;
Finalize the instance
function Compute(const DataNameValue: array of const; const Issuer: RawUTF8=''; const Subject: RawUTF8=''; const Audience: RawUTF8=''; NotBefore: TDateTime=0; ExpirationMinutes: integer=0; Signature: PRawUTF8=nil): RawUTF8;

Compute a new JWT for a given payload
- here the data payload is supplied as Name,Value pairs - by convention, some registered Names (see TJWTClaim) should not be used here, and private claims names are expected to be short (typically 3 chars), or an URI
- depending on the instance Claims, you should also specify associated Issuer, Subject, Audience and NotBefore values; expected 'exp', 'nbf', 'iat', 'jti' claims will also be generated and included, if needed
- you can override the aExpirationMinutes value as defined in Create()
- Audience is usually a single text, serialized as a JSON string, but if the value supplied starts with '[' , it is expected to be an array of text values, already serialized as a JSON array of strings
- this method is thread-safe

function ComputeAuthorizationHeader(const DataNameValue: array of const; const Issuer: RawUTF8=''; const Subject: RawUTF8=''; const Audience: RawUTF8=''; NotBefore: TDateTime=0; ExpirationMinutes: integer=0): RawUTF8;

Compute a HTTP Authorization header containing a JWT for a given payload
- just a wrapper around Compute(), returned the HTTP header value:
  Authorization: <HttpAuthorizationHeader>

- this method is thread-safe

function Verify(const Token: RawUTF8): TJWTResult; overload;

Check a JWT value, and its signature
- will validate all expected Claims, and the associated signature
- verification state is returned as function result
- supplied JWT is transmitted e.g. in HTTP header:
  Authorization: Bearer <Token>

- this method is thread-safe

function VerifyAuthorizationHeader(const HttpAuthorizationHeader: RawUTF8; out JWT: TJWTContent): boolean; overload;

Check a HTTP Authorization header value as JWT, and its signature
- will validate all expected Claims, and the associated signature
- verification state is returned in JWT.result (jwtValid for a valid JWT), together with all parsed payload information
- expect supplied HttpAuthorizationHeader as transmitted in HTTP header:
  Authorization: <HttpAuthorizationHeader>

- this method is thread-safe
class function VerifyPayload(const Token, ExpectedSubject, ExpectedIssuer, ExpectedAudience: RawUTF8; Expiration: PUnixTime=nil; Signature: PRawUTF8=nil; Payload: PVariant=nil; IgnoreTime: boolean=false; NotBeforeDelta: TUnixTime=15): TJWTResult;

In-place decoding and quick check of the JWT payload
- it won't check the signature, but the header's algorithm against the class name (use TJWTAbstract class to allow any algorithm)
- it will decode the JWT payload and check for its expiration, and some mandatory field values - you can optionally retrieve the Expiration time, the ending Signature, and/or the Payload decoded as TDocVariant
- NotBeforeDelta allows to define some time frame for the "nbf" field
- may be used on client side to quickly validate a JWT received from server, without knowing the exact algorithm or secret keys

procedure Verify(const Token: RawUTF8; out JWT: TJWTContent; ExcludedClaims: TJWTClaims=[]); overload;

Check a JWT value, and its signature
- will validate all expected Claims (minus ExcludedClaims optional parameter), and the associated signature
- verification state is returned in JWT.result (jwtValid for a valid JWT), together with all parsed payload information
- supplied JWT is transmitted e.g. in HTTP header:
  Authorization: Bearer <Token>
- this method is thread-safe

property Algorithm: RawUTF8 read fAlgorithm;
The name of the algorithm used by this instance (e.g. 'HS256')

property Audience: TRawUTF8DynArray read fAudience;
The audience string values associated with this instance
- will be checked by Verify() method, and set in TJWTContent.audience

property CacheResults: TJWTResults read fCacheResults write fCacheResults;
Which TJWTContent.result should be stored in in-memory cache
- default is [jwtValid] but you may also include jwtInvalidSignature if signature checking uses a lot of resources
- only used if CacheTimeoutSeconds>0

property CacheTimeoutSeconds: integer read fCacheTimeoutSeconds write SetCacheTimeoutSeconds;
Delay of optional in-memory cache of Verify() TJWTContent
- equals 0 by default, i.e. cache is disabled
- may be useful if the signature process is very resource consuming (e.g. for TJWTES256 or even HMAC-SHA-256) - see also CacheResults
- each time this property is assigned, internal cache content is flushed

property Claims: TJWTClaims read fClaims;
The JWT Registered Claims, as implemented by this instance
- Verify() method will ensure all claims are defined in the payload, then fill TJWTContent.reg[] with all corresponding values

property ExpirationSeconds: integer read fExpirationSeconds;
The period, in seconds, for the "exp" claim
**property** Options: TJWTOptions **read** fOptions **write** fOptions;

Allow to tune the Verify and Compute method process

**TJWTNone** = **class**(TJWTAbstract)

*Implements JSON Web Tokens using 'none' algorithm*
- you should never use this weak algorithm in production, unless your communication is already secured by other means, and use JWT as cookies

**constructor** Create(aClaims: TJWTClaims; **const** aAudience: **array** of RawUTF8;
aExpirationMinutes: integer=0; aIDIdentifier: TSynUniqueIdentifierProcess=0;
aIDObfuscationKey: RawUTF8=''); **reintroduce**;

*Initialize the JWT processing using the 'none' algorithm*
- the supplied set of claims are expected to be defined in the JWT payload
- aAudience are the allowed values for the jrcAudience claim
- aExpirationMinutes is the deprecation time for the jrcExpirationTime claim
- aIDIdentifier and aIDObfuscationKey are passed to a TSynUniqueIdentifierGenerator instance used for jrcJwtID claim

**TJWTSynSignerAbstract** = **class**(TJWTAbstract)

*Abstract parent of JSON Web Tokens using HMAC-SHA2 or SHA-3 algorithms*
- SHA-3 is not yet officially defined in [@http://tools.ietf.org/html/rfc7518 but could be used as a safer (and sometimes faster) alternative to HMAC-SHA2](http://tools.ietf.org/html/rfc7518)
- digital signature will be processed by an internal TSynSigner instance
- never use this abstract class, but any inherited class, or JWT_CLASS[].Create to instantiate a JWT process from a given algorithm

**constructor** Create(**const** aSecret: RawUTF8; aSecretPBKDF2Rounds: integer; aClaims: TJWTClaims; **const** aAudience: **array** of RawUTF8; aExpirationMinutes: integer=0; aIDIdentifier: TSynUniqueIdentifierProcess=0;
aIDObfuscationKey: RawUTF8=''; aPBKDF2Secret: PHash512Rec=nil); **reintroduce**;

*Initialize the JWT processing using SHA3 algorithm*
- the supplied set of claims are expected to be defined in the JWT payload
- the supplied secret text will be used to compute the digital signature, directly if aSecretPBKDF2Rounds=0, or via PBKDF2 iterative key derivation if some number of rounds were specified
- aAudience are the allowed values for the jrcAudience claim
- aExpirationMinutes is the deprecation time for the jrcExpirationTime claim
- aIDIdentifier and aIDObfuscationKey are passed to a TSynUniqueIdentifierGenerator instance used for jrcJwtID claim
- optionally return the PBKDF2 derivated key for aSecretPBKDF2Rounds>0

**destructor** Destroy; **override**;

*Finalize the instance*

**property** SignatureAlgo: TSignAlgo **read** fSignPrepared.fAlgo;

*The TSynSigner raw algorithm used for digital signature*

**property** SignatureSize: integer **read** fSignPrepared.fSignatureSize;

*The digital signature size, in byte*
property SignPrepared: TSynSigner; read fSignPrepared;
Low-level read access to the internal signature structure

TJWTHS256 = class(TJWTSynSignerAbstract)
  Implements JSON Web Tokens using ‘HS256’ (HMAC SHA-256) algorithm
  - as defined in @http://tools.ietf.org/html/rfc7518 paragraph 3.2
  - our HMAC SHA-256 implementation used is thread safe, and very fast (x86: 3us, x64: 2.5us) so
    cache is not needed
  - resulting signature size will be of 256 bits

TJWTHS384 = class(TJWTSynSignerAbstract)
  Implements JSON Web Tokens using ‘HS384’ (HMAC SHA-384) algorithm
  - as defined in @http://tools.ietf.org/html/rfc7518 paragraph 3.2
  - our HMAC SHA-384 implementation used is thread safe, and very fast even on x86 (if the CPU
    supports SSE3 opcodes)
  - resulting signature size will be of 384 bits

TJWTHS512 = class(TJWTSynSignerAbstract)
  Implements JSON Web Tokens using ‘HS512’ (HMAC SHA-512) algorithm
  - as defined in @http://tools.ietf.org/html/rfc7518 paragraph 3.2
  - our HMAC SHA-512 implementation used is thread safe, and very fast even on x86 (if the CPU
    supports SSE3 opcodes)
  - resulting signature size will be of 512 bits

TJWTS3224 = class(TJWTSynSignerAbstract)
  Experimental JSON Web Tokens using SHA3-224 algorithm
  - SHA-3 is not yet officially defined in @http://tools.ietf.org/html/rfc7518 but could be used as a
    safer (and sometimes faster) alternative to HMAC-SHA2
  - resulting signature size will be of 224 bits

TJWTS3256 = class(TJWTSynSignerAbstract)
  Experimental JSON Web Tokens using SHA3-256 algorithm
  - SHA-3 is not yet officially defined in @http://tools.ietf.org/html/rfc7518 but could be used as a
    safer (and sometimes faster) alternative to HMAC-SHA2
  - resulting signature size will be of 256 bits

TJWTS3384 = class(TJWTSynSignerAbstract)
  Experimental JSON Web Tokens using SHA3-384 algorithm
  - SHA-3 is not yet officially defined in @http://tools.ietf.org/html/rfc7518 but could be used as a
    safer (and sometimes faster) alternative to HMAC-SHA2
  - resulting signature size will be of 384 bits

TJWTS3512 = class(TJWTSynSignerAbstract)
  Experimental JSON Web Tokens using SHA3-512 algorithm
  - SHA-3 is not yet officially defined in @http://tools.ietf.org/html/rfc7518 but could be used as a
    safer (and sometimes faster) alternative to HMAC-SHA2
  - resulting signature size will be of 512 bits
 TJWTS3S128 = class(TJWTSynSignerAbstract)

 Experimental JSON Web Tokens using SHA3-SHAKE128 algorithm
 - SHA-3 is not yet officially defined in @http://tools.ietf.org/html/rfc7518 but could be used as a safer (and sometimes faster) alternative to HMAC-SHA2
 - resulting signature size will be of 256 bits

 TJWTS3S256 = class(TJWTSynSignerAbstract)

 Experimental JSON Web Tokens using SHA3-SHAKE256 algorithm
 - SHA-3 is not yet officially defined in @http://tools.ietf.org/html/rfc7518 but could be used as a safer (and sometimes faster) alternative to HMAC-SHA2
 - resulting signature size will be of 512 bits

 Types implemented in the SynCrypto unit

 IProtocolDynArray = array of IProtocol;
 Stores a list of IProtocol instances

 PAESBlock = ^TAESBlock;
 Points to a 128 bits memory block, as used for AES data cypher/uncypher

 PHMAC_CRC32C = ^THMAC_CRC32C;
 Points to HMAC message authentication code using crc32c as hash function

 PHMAC_SHA1 = ^THMAC_SHA1;
 Points to a HMAC message authentication context using SHA-1

 PHMAC_SHA256 = ^THMAC_SHA256;
 Points to a HMAC message authentication context using SHA-256

 PHMAC_SHA384 = ^THMAC_SHA384;
 Points to a HMAC message authentication context using SHA-384

 PHMAC_SHA512 = ^THMAC_SHA512;
 Points to a HMAC message authentication context using SHA-512

 PJWTContent = ^TJWTContent;
 Pointer to a JWT decoded content, as processed by TJWTAbstract

 PSHA384 = ^TSHA384;
 Points to SHA-384 hashing instance

 PSHA512 = ^TSHA512;
 Points to SHA-512 hashing instance

 PSynSigner = ^TSynSigner;
 Reference to TSynSigner wrapper object

 Short32 = string[32];
 32-characters ASCII string, e.g. as returned by AESBlockToShortString()

 TAESAbstractClass = class of TAESAbstract;
 Class-reference type (metaclass) of an AES cypher/uncypher

 TAESBlock = THash128;
 128 bits memory block for AES data cypher/uncypher
TAESIVReplayAttackCheck = ( repNoCheck, repCheckedIfAvailable, repMandatory );

How TAESAbstract.DecryptPKCS7 should detect replay attack
- repNoCheck and repCheckedIfAvailable will be compatible with older versions of the protocol, but repMandatory will reject any encryption without the TAESIVCTR algorithm

TAESKey = THash256;
256 bits memory block for maximum AES key storage

THashAlgo = ( hfMD5, hfSHA1, hfSHA256, hfSHA384, hfSHA512, hfSHA3_256, hfSHA3_512 );
Hash algorithms available for HashFile/HashFull functions and TSynHasher object

THashAlgos = set of THashAlgo;
Set of algorithms available for HashFile/HashFull functions and TSynHasher object

TJWTAbstractClass = class of TJWTAbstract;
Class-reference type (metaclass) of a JWT algorithm process

TJWTClaims = set of TJWTClaim;
Set of JWT Registered Claims, as defined in RFC 7519
- known registered claims have a specific name and behavior, and will be handled automatically by TJWTAbstract
- corresponding field names are iss, sub, aud, exp, nbf, iat, jti - as defined in JWT_CLAIMS_TEXT constant
- jrcIssuer identifies the server which originated the token, e.g. "iss": "https://example.auth0.com/" when the token comes from Auth0 servers
- jrcSubject is the application-specific extent which is protected by this JWT, e.g. an User or Resource ID, e.g. "sub": "auth0|57fe9f1bad961aa242870e"
- jrcAudience claims that the token is valid only for one or several resource servers (may be a JSON string or a JSON array of strings), e.g. "aud":["https://myshineyfileserver.sometld"] - TJWTAbstract will check that the supplied "aud" field does match an expected list of identifiers
- jrcExpirationTime contains the Unix timestamp in seconds after which the token must not be granted access, e.g. "exp":1477474667
- jrcNotBefore contains the Unix timestamp in seconds before which the token must not be granted access, e.g. "nbf":147745438
- jrcIssuedAt contains the Unix timestamp in seconds when the token was generated, e.g. "iat":1477438667
- jrcJwtID provides a unique identifier for the JWT, to prevent any replay; TJWTAbstract.Compute will set an obfuscated TSynUniqueIdentifierGenerator hexadecimal value

TJWTClaims = set of TJWTClaim;
Set of JWT Registered Claims, as in TJWTAbstract.Claims

TJWTContentDynArray = array of TJWTContent;
Used to store a list of JWT decoded content
- as used e.g. by TJWTAbstract cache

TJWTOption = ( joHeaderParse, joAllowUnexpectedClaims, joAllowUnexpectedAudience, joNoJwtIDGenerate, joNoJwtIDCheck, joDoubleInData );
Available options for TJWTAbstract process

TJWTOptions = set of TJWTOption;
Store options for TJWTAbstract process

TJWTResult = ( jwtValid, jwtNoToken, jwtWrongFormat, jwtInvalidAlgorithm,
jwtInvalidPayload, jwtUnexpectedClaim, jwtMissingClaim, jwtUnknownAudience, 
jwtExpired, jwtNotBeforeFailed, jwtInvalidIssuedAt, jwtInvalidID, jwtInvalidSignature);

TJWTContent.result codes after TJWTAbstract.Verify method call

TJWTResults = set of TJWTResult;

// set of TJWTContent.result codes

TJWTSynSignerAbstractClass = class of TJWTSynSignerAbstract;

Meta-class for TJWTSynSignerAbstract creations

TMD5Digest = THash128;

128 bits memory block for MD5 hash digest storage

TProtocolAESClass = class of TProtocolAES;

Class-reference type (metaclass) of an AES secure protocol

TProtocolResult = (sprSuccess, sprBadRequest, sprUnsupported, sprUnexpectedAlgorithm, 
sprInvalidCertificate, sprInvalidSignature, sprInvalidEphemeralKey, 
sprInvalidPublicKey, sprInvalidPrivateKey, sprInvalidMAC );

Possible return codes by IProtocol classes

TSHA1Digest = THash160;

160 bits memory block for SHA-1 hash digest storage

TSHA256Digest = THash256;

256 bits (32 bytes) memory block for SHA-256 hash digest storage

TSHA384Digest = THash384;

384 bits (64 bytes) memory block for SHA-384 hash digest storage

TSHA3Algo = ( SHA3_224, SHA3_256, SHA3_384, SHA3_512, SHAKE_128, SHAKE_256 );

SHA-3 instances, as defined by NIST Standard for Keccak sponge construction

TSHA512Digest = THash512;

512 bits (64 bytes) memory block for SHA-512 hash digest storage

TSignAlgo = ( saSha1, saSha256, saSha384, saSha512, saSha3224, saSha3256, saSha3384, 
saSha3512, saSha35128, saSha35256 );

The HMAC/SHA-3 algorithms known by TSynSigner

Constants implemented in the SynCrypto unit

AESBlockMod = 15;

Bit mask for fast modulo of AES block size

AESBlockShift = 4;

Power of two for a standard AES block size during cypher/uncypher
- to be used as 1 shl AESBlockShift or 1 shr AESBlockShift for fast div/mod

AESContextSize = 276+sizeof(pointer) +sizeof(pointer);

Hide all AES Context complex code

AESKeySize = 256 div 8;

Maximum AES key size (in bytes)

JWT_CLAIMS_TEXT: array[TJWTClaim] of RawUTF8 = (}
The text field names of the registered claims, as defined by RFC 7519
- see TJWTClaim enumeration and TJWTClaims set
- RFC standard expects those to be case-sensitive

```
iss', 'sub', 'aud', 'exp', 'nbf', 'iat', 'jti');
```

*JWT_CLASS:* array[TSignAlgo] of TJWTSynSignerAbstractClass = (TJWTHS256, TJWTHS256, TJWTHS384, TJWTHS512, TJWTS3224, TJWTS3256, TJWTS3384, TJWTS3512, TJWTS3S128, TJWTS3S256);

Able to instantiate any of the TJWTSynSignerAbstract instance expected
- SHA-1 will fallback to TJWTHS256 (since SHA-1 will never be supported)
- SHA-3 is not yet officially defined in @http://tools.ietf.org/html/rfc7518
- typical use is the following:
  result := JWT_CLASS[algo].Create(master, round, claims, [], expirationMinutes);

```
JWT_TEXT: array[TSignAlgo] of RawUTF8 = ('HS256', 'HS256', 'HS384', 'HS512', 'S3224', 'S3256', 'S3384', 'S3512', 'S3S128', 'S3S256');
```

How TJWTSynSignerAbstract algorithms are identified in the JWT
- SHA-1 will fallback to HS256 (since there will never be SHA-1 support)
- SHA-3 is not yet officially defined in @http://tools.ietf.org/html/rfc7518

SHA3ContextSize = 412;

Hide all SHA-3 complex code by storing the Keccak Sponge as buffer

SHAContextSize = 108;

Hide all SHA-1/SHA-2 complex code by storing the context as buffer

**Functions or procedures implemented in the SynCrypto unit**

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adler32Asm</td>
<td>Fast Adler32 implementation</td>
<td>1189</td>
</tr>
<tr>
<td>Adler32Pas</td>
<td>Simple Adler32 implementation</td>
<td>1189</td>
</tr>
<tr>
<td>Adler32SelfTest</td>
<td>Self test of Adler32 routines</td>
<td>1189</td>
</tr>
<tr>
<td>AES</td>
<td>Direct Encrypt/Decrypt of data using the TAES class</td>
<td>1190</td>
</tr>
<tr>
<td>AES</td>
<td>Direct Encrypt/Decrypt of data using the TAES class</td>
<td>1190</td>
</tr>
<tr>
<td>AES</td>
<td>Direct Encrypt/Decrypt of data using the TAES class</td>
<td>1190</td>
</tr>
<tr>
<td>AES</td>
<td>Direct Encrypt/Decrypt of data using the TAES class</td>
<td>1190</td>
</tr>
<tr>
<td>AES</td>
<td>Direct Encrypt/Decrypt of data using the TAES class</td>
<td>1190</td>
</tr>
<tr>
<td>AESBlockToShortString</td>
<td>Compute the hexadecimal representation of an AES 16-byte block</td>
<td>1190</td>
</tr>
<tr>
<td>AESBlockToShortString</td>
<td>Compute the hexadecimal representation of an AES 16-byte block</td>
<td>1190</td>
</tr>
<tr>
<td>AESBlockToString</td>
<td>Compute the hexadecimal representation of an AES 16-byte block</td>
<td>1190</td>
</tr>
<tr>
<td>AESFull</td>
<td>AES and XOR encryption using the TAESFull format</td>
<td>1190</td>
</tr>
<tr>
<td>AESFull</td>
<td>AES and XOR encryption using the TAESFull format</td>
<td>1190</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>AESFullKeyOK</td>
<td>AES and XOR decryption check using the TAESFull format</td>
<td>1190</td>
</tr>
<tr>
<td>AESIVCtrEncryptDecrypt</td>
<td>Global shared function which may encrypt or decrypt any 128-bit block using AES-128 and the global AESIVCTR_KEY</td>
<td>1190</td>
</tr>
<tr>
<td>AESSelfTest</td>
<td>Self test of AES routines</td>
<td>1190</td>
</tr>
<tr>
<td>AESSHA256</td>
<td>AES encryption using the TAES format with a supplied SHA-256 password</td>
<td>1191</td>
</tr>
<tr>
<td>AESSHA256</td>
<td>AES encryption using the TAES format with a supplied SHA-256 password</td>
<td>1191</td>
</tr>
<tr>
<td>AESSHA256</td>
<td>AES encryption using the TAES format with a supplied SHA-256 password</td>
<td>1191</td>
</tr>
<tr>
<td>AESSHA256Full</td>
<td>AES encryption using the TAESFull format with a supplied SHA-256 password</td>
<td>1191</td>
</tr>
<tr>
<td>AFDiffusion</td>
<td>Low-level anti-forensic diffusion of a memory buffer using SHA-256</td>
<td>1191</td>
</tr>
<tr>
<td>bswap160</td>
<td>Little endian fast conversion</td>
<td>1191</td>
</tr>
<tr>
<td>bswap256</td>
<td>Little endian fast conversion</td>
<td>1191</td>
</tr>
<tr>
<td>CompressShaAes</td>
<td>Encrypt data content using the AES-256/CFB algorithm, after SynLZ compression</td>
<td>1191</td>
</tr>
<tr>
<td>CompressShaAesSetKey</td>
<td>Set a text-based encryption key for CompressShaAes() global function</td>
<td>1191</td>
</tr>
<tr>
<td>CryptDataForCurrentUser</td>
<td>Protect some data via AES-256-CFB and a secret known by the current user only</td>
<td>1192</td>
</tr>
<tr>
<td>CryptDataForCurrentUserDPAPI</td>
<td>Protect some data for the current user, using Windows DPAPI</td>
<td>1192</td>
</tr>
<tr>
<td>FillSystemRandom</td>
<td>Low-level function returning some random binary using standard API</td>
<td>1192</td>
</tr>
<tr>
<td>Hash128ToDouble</td>
<td>Low-level function able to derive a 0..1 64-bit floating-point from 128-bit of data</td>
<td>1193</td>
</tr>
<tr>
<td>Hash128ToExt</td>
<td>Low-level function able to derive a 0..1 floating-point from 128-bit of data</td>
<td>1193</td>
</tr>
<tr>
<td>Hash128ToSingle</td>
<td>Low-level function able to derive a 0..1 32-bit floating-point from 128-bit of data</td>
<td>1193</td>
</tr>
<tr>
<td>HashFile</td>
<td>Compute the hexadecimal hash of any (big) file</td>
<td>1193</td>
</tr>
<tr>
<td>HashFile</td>
<td>Compute the hexadecimal hash(es) of one file, as external .md5/.sha256/... files</td>
<td>1193</td>
</tr>
<tr>
<td>HashFull</td>
<td>One-step hash computation of a buffer as lowercase hexadecimal string</td>
<td>1193</td>
</tr>
<tr>
<td>HMAC_CRC256C</td>
<td>Compute the HMAC message authentication code using crc256c as hash function</td>
<td>1193</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>HMAC_CRC256C</td>
<td>Compute the HMAC message authentication code using crc256c as hash function</td>
<td>1193</td>
</tr>
<tr>
<td>HMAC_CRC256C</td>
<td>Compute the HMAC message authentication code using crc256c as hash function</td>
<td>1193</td>
</tr>
<tr>
<td>HMAC_CRC32C</td>
<td>Compute the HMAC message authentication code using crc32c as hash function</td>
<td>1194</td>
</tr>
<tr>
<td>HMAC_CRC32C</td>
<td>Compute the HMAC message authentication code using crc32c as hash function</td>
<td>1194</td>
</tr>
<tr>
<td>HMAC_CRC32C</td>
<td>Compute the HMAC message authentication code using crc32c as hash function</td>
<td>1194</td>
</tr>
<tr>
<td>HMAC_SHA1</td>
<td>Compute the HMAC message authentication code using SHA-1 as hash function</td>
<td>1194</td>
</tr>
<tr>
<td>HMAC_SHA1</td>
<td>Compute the HMAC message authentication code using SHA-1 as hash function</td>
<td>1194</td>
</tr>
<tr>
<td>HMAC_SHA1</td>
<td>Compute the HMAC message authentication code using SHA-1 as hash function</td>
<td>1194</td>
</tr>
<tr>
<td>HMAC_SHA256</td>
<td>Compute the HMAC message authentication code using SHA-256 as hash function</td>
<td>1194</td>
</tr>
<tr>
<td>HMAC_SHA256</td>
<td>Compute the HMAC message authentication code using SHA-256 as hash function</td>
<td>1194</td>
</tr>
<tr>
<td>HMAC_SHA256</td>
<td>Compute the HMAC message authentication code using SHA-256 as hash function</td>
<td>1194</td>
</tr>
<tr>
<td>HMAC_SHA384</td>
<td>Compute the HMAC message authentication code using SHA-384 as hash function</td>
<td>1194</td>
</tr>
<tr>
<td>HMAC_SHA384</td>
<td>Compute the HMAC message authentication code using SHA-384 as hash function</td>
<td>1194</td>
</tr>
<tr>
<td>HMAC_SHA384</td>
<td>Compute the HMAC message authentication code using SHA-384 as hash function</td>
<td>1194</td>
</tr>
<tr>
<td>HMAC_SHA512</td>
<td>Compute the HMAC message authentication code using SHA-512 as hash function</td>
<td>1195</td>
</tr>
<tr>
<td>HMAC_SHA512</td>
<td>Compute the HMAC message authentication code using SHA-512 as hash function</td>
<td>1195</td>
</tr>
<tr>
<td>HMAC_SHA512</td>
<td>Compute the HMAC message authentication code using SHA-512 as hash function</td>
<td>1195</td>
</tr>
<tr>
<td>htdigest</td>
<td>Compute the HTDigest for a user and a realm, according to a supplied password</td>
<td>1195</td>
</tr>
<tr>
<td>MD5</td>
<td>Direct MD5 hash calculation of some data (string-encoded)</td>
<td>1195</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>MD5Buf</td>
<td>Direct MD5 hash calculation of some data</td>
<td>1195</td>
</tr>
<tr>
<td>MD5DigestToString</td>
<td>Compute the hexadecimal representation of a MD5 digest</td>
<td>1195</td>
</tr>
<tr>
<td>MD5SelfTest</td>
<td>Self test of MD5 routines</td>
<td>1195</td>
</tr>
<tr>
<td>MD5StringToDigest</td>
<td>Compute the MD5 digest from its hexadecimal representation</td>
<td>1195</td>
</tr>
<tr>
<td>PBKDF2_HMAC_SHA1</td>
<td>Compute the PBKDF2 derivation of a password using HMAC over SHA-1</td>
<td>1195</td>
</tr>
<tr>
<td>PBKDF2_HMAC_SHA256</td>
<td>Compute the PBKDF2 derivation of a password using HMAC over SHA-256</td>
<td>1195</td>
</tr>
<tr>
<td>PBKDF2_HMAC_SHA256</td>
<td>Compute the PBKDF2 derivation of a password using HMAC over SHA-256, into several 256-bit items, so can be used to return any size of output key</td>
<td>1195</td>
</tr>
<tr>
<td>PBKDF2_HMAC_SHA384</td>
<td>Compute the PBKDF2 derivation of a password using HMAC over SHA-384</td>
<td>1195</td>
</tr>
<tr>
<td>PBKDF2_HMAC_SHA512</td>
<td>Compute the PBKDF2 derivation of a password using HMAC over SHA-512</td>
<td>1195</td>
</tr>
<tr>
<td>PBKDF2_SHA3</td>
<td>Safe key derivation using iterated SHA-3 hashing</td>
<td>1196</td>
</tr>
<tr>
<td>PBKDF2_SHA3_Crypt</td>
<td>Encryption/decryption of any data using iterated SHA-3 hashing key derivation</td>
<td>1196</td>
</tr>
<tr>
<td>RawMd5Compress</td>
<td>Entry point of the raw MD5 transform function - may be used for low-level use</td>
<td>1196</td>
</tr>
<tr>
<td>RawSha1Compress</td>
<td>Entry point of the raw SHA-1 transform function - may be used for low-level use</td>
<td>1196</td>
</tr>
<tr>
<td>RawSha256Compress</td>
<td>Entry point of the raw SHA-256 transform function - may be used for low-level use</td>
<td>1196</td>
</tr>
<tr>
<td>RawSha512Compress</td>
<td>Entry point of the raw SHA-512 transform function - may be used for low-level use</td>
<td>1196</td>
</tr>
<tr>
<td>RC4SelfTest</td>
<td>Self test of RC4 routines</td>
<td>1196</td>
</tr>
<tr>
<td>SHA1</td>
<td>Direct SHA-1 hash calculation of some data (string-encoded)</td>
<td>1196</td>
</tr>
<tr>
<td>SHA1DigestToString</td>
<td>Compute the hexadecimal representation of a SHA-1 digest</td>
<td>1196</td>
</tr>
<tr>
<td>SHA1SelfTest</td>
<td>Self test of SHA-1 routines</td>
<td>1196</td>
</tr>
<tr>
<td>SHA1StringToDigest</td>
<td>Compute the SHA-1 digest from its hexadecimal representation</td>
<td>1196</td>
</tr>
<tr>
<td>SHA256</td>
<td>Direct SHA-256 hash calculation of some data (string-encoded)</td>
<td>1196</td>
</tr>
<tr>
<td>SHA256</td>
<td>Direct SHA-256 hash calculation of some binary data</td>
<td>1196</td>
</tr>
<tr>
<td>SHA256Digest</td>
<td>Direct SHA-256 hash calculation of some binary data</td>
<td>1197</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>SHA256Digest</td>
<td>Direct SHA-256 hash calculation of some binary data</td>
<td>1197</td>
</tr>
<tr>
<td>SHA256DigestToString</td>
<td>Compute the hexadecimal representation of a SHA-256 digest</td>
<td>1197</td>
</tr>
<tr>
<td>SHA256SelfTest</td>
<td>Self test of SHA-256 routines</td>
<td>1197</td>
</tr>
<tr>
<td>SHA256StringToDigest</td>
<td>Compute the SHA-256 digest from its hexadecimal representation</td>
<td>1197</td>
</tr>
<tr>
<td>SHA256Weak</td>
<td>Direct SHA-256 hash calculation of some data (string-encoded)</td>
<td>1197</td>
</tr>
<tr>
<td>SHA3</td>
<td>Direct SHA-3 hash calculation of some binary buffer</td>
<td>1197</td>
</tr>
<tr>
<td>SHA3</td>
<td>Direct SHA-3 hash calculation of some data (string-encoded)</td>
<td>1197</td>
</tr>
<tr>
<td>SHA384</td>
<td>Direct SHA-384 hash calculation of some data (string-encoded)</td>
<td>1197</td>
</tr>
<tr>
<td>SHA384DigestToString</td>
<td>Compute the hexadecimal representation of a SHA-384 digest</td>
<td>1198</td>
</tr>
<tr>
<td>SHA512</td>
<td>Direct SHA-512 hash calculation of some data (string-encoded)</td>
<td>1198</td>
</tr>
<tr>
<td>SHA512DigestToString</td>
<td>Compute the hexadecimal representation of a SHA-512 digest</td>
<td>1198</td>
</tr>
<tr>
<td>XorBlock</td>
<td>Apply the XOR operation to the supplied binary buffers of 16 bytes</td>
<td>1198</td>
</tr>
<tr>
<td>XorBlock16</td>
<td>Apply the XOR operation to the supplied binary buffers of 16 bytes</td>
<td>1198</td>
</tr>
<tr>
<td>XorConst</td>
<td>Fast XOR Cypher changing by Count value</td>
<td>1198</td>
</tr>
<tr>
<td>XorOffset</td>
<td>Fast and simple XOR Cypher using Index (=Position in Dest Stream)</td>
<td>1198</td>
</tr>
</tbody>
</table>

**Function Adler32Asm(Adler: cardinal; p: pointer; Count: Integer): cardinal;**
- Fast Adler32 implementation
- 16-bytes-chunk unrolled asm version

**Function Adler32Pas(Adler: cardinal; p: pointer; Count: Integer): cardinal;**
- Simple Adler32 implementation
- a bit slower than Adler32Asm() version below, but shorter code size

**Function Adler32SelfTest: boolean;**
- Self test of Adler32 routines

**Function AES(const Key: cardinal; KeySize: cardinal; const s: RawByteString; Encrypt: boolean): RawByteString; overload;**
- Direct Encrypt/Decrypt of data using the TAES class
- last bytes (not part of 16 bytes blocks) are not encrypted by AES, but with XOR

**Function AES(const Key: cardinal; KeySize: cardinal; buffer: pointer; Len: cardinal; Stream: TStream; Encrypt: boolean): boolean; overload;**
- Direct Encrypt/Decrypt of data using the TAES class
- last bytes (not part of 16 bytes blocks) are not encrypted by AES, but with XOR
procedure AES(const Key; KeySize: cardinal; buffer: pointer; Len: Integer; Encrypt: boolean); overload;

Direct Encrypt/Decrypt of data using the TAES class
- last bytes (not part of 16 bytes blocks) are not crypted by AES, but with XOR

procedure AES(const Key; KeySize: cardinal; bIn, bOut: pointer; Len: Integer; Encrypt: boolean); overload;

Direct Encrypt/Decrypt of data using the TAES class
- last bytes (not part of 16 bytes blocks) are not crypted by AES, but with XOR

function AESBlockToShortString(const block: TAESBlock): short32; overload;

Compute the hexadecimal representation of an AES 16-byte block
- returns a stack-allocated short string

procedure AESBlockToShortString(const block: TAESBlock; out result: short32); overload;

Compute the hexadecimal representation of an AES 16-byte block
- fill a stack-allocated short string

function AESBlockToString(const block: TAESBlock): RawUTF8;

Compute the hexadecimal representation of an AES 16-byte block

function AESFull(const Key; KeySize: cardinal; bIn: pointer; Len: Integer; outStream: TStream; Encrypt: boolean; OriginalLen: Cardinal=0): boolean; overload;

AES and XOR encryption using the TAESFull format
- outStream will be larger/smaller than Len (full AES encrypted)
- returns true if OK

function AESFull(const Key; KeySize: cardinal; bIn, bOut: pointer; Len: Integer; Encrypt: boolean; OriginalLen: Cardinal=0): integer; overload;

AES and XOR encryption using the TAESFull format
- bOut must be at least bIn+32/Encrypt bIn-16/Decrypt
- returns outLength, -1 if error

function AESFullKeyOK(const Key; KeySize: cardinal; buff: pointer): boolean;

AES and XOR decryption check using the TAESFull format
- return true if beginning of buff contains true AESFull encrypted data with this Key
- if not KeySize in [128,192,256] -> use fast and efficient Xor Cypher

procedure AESIVCtrEncryptDecrypt(const BI; var BO; DoEncrypt: boolean);

Global shared function which may encrypt or decrypt any 128-bit block using AES-128 and the global AESIVCTR_KEY

function AESSelfTest(onlytables: Boolean): boolean;

Self test of AES routines

procedure AESSHA256(Buffer: pointer; Len: integer; const Password: RawByteString; Encrypt: boolean); overload;

AES encryption using the TAES format with a supplied SHA-256 password
- last bytes (not part of 16 bytes blocks) are not crypted by AES, but with XOR
procedure AESSHA256(bIn, bOut: pointer; Len: integer; const Password: RawByteString; Encrypt: boolean); overload;

- AES encryption using the TAES format with a supplied SHA-256 password
- last bytes (not part of 16 bytes blocks) are not crypted by AES, but with XOR

function AESSHA256(const s, Password: RawByteString; Encrypt: boolean): RawByteString; overload;

- AES encryption using the TAES format with a supplied SHA-256 password
- last bytes (not part of 16 bytes blocks) are not crypted by AES, but with XOR

procedure AESSHA256Full(bIn: pointer; Len: Integer; outStream: TStream; const Password: RawByteString; Encrypt: boolean); overload;

- AES encryption using the TAESFull format with a supplied SHA-256 password
- outStream will be larger/smaller than Len: this is a full AES version with a triming TAESFullHeader at the beginning

procedure AFDiffusion(buf, rnd: pointer; size: cardinal);

- Low-level anti-forensic diffusion of a memory buffer using SHA-256
- as used by TAESPRNG.AFSplit and TAESPRNG.AFUnSplit

procedure bswap160(s, d: PIntegerArray);

- Little endian fast conversion
- 160 bits = 5 integers
- use fast bswap asm in x86/x64 mode

procedure bswap256(s, d: PIntegerArray);

- Little endian fast conversion
- 256 bits = 8 integers
- use fast bswap asm in x86/x64 mode

function CompressShaAes(var DataRawByteString; Compress: boolean): AnsiString;

- Encrypt data content using the AES-256/CFB algorithm, after SynLZ compression
- as expected by THttpSocket.RegisterCompress()
- will return 'synshaaes' as ACCEPT-ENCODING: header parameter
- will use global CompressShaAesKey / CompressShaAesClass variables to be set according to the expected algorithm and Key e.g. via a call to CompressShaAesSetKey()
- if you want to change the chaining mode, you can customize the global CompressShaAesClass variable to the expected TAES* class name
- will store a hash of both cyphered and clear stream: if the data is corrupted during transmission, will instantly return "

procedure CompressShaAesSetKey(const Key: RawByteString; AesClass: TAESAbstractClass=nil);

- Set an text-based encryption key for CompressShaAes() global function
- will compute the key via SHA256Weak() and set CompressShaAesKey
- the key is global to the whole process
function CryptDataForCurrentUser(const Data, AppSecret: RawByteString; Encrypt: boolean): RawByteString;

Protect some data via AES-256-CFB and a secret known by the current user only
- the application can specify a secret salt text, which should reflect the current execution context,
  to ensure nobody could decrypt the data without knowing this application-specific AppSecret value
- here data is cyphered using a random secret key, stored in a file located in
  GetSystemPath(spUserData)+sep+PBKDF2_HMAC_SHA256(CryptProtectDataEntropy,User)
with sep='_' under Windows, and sep='.' under Linux/Posix
- under Windows, it will encode the secret file via CryptProtectData DPAPI, so has the same
  security level than plain CryptDataForCurrentUserDPAPI()
- under Linux/POSIX, access to the $HOME user's .xxxxxxxxxx secret file with chmod 400 is
  considered to be a safe enough approach
- this function is up to 100 times faster than CryptDataForCurrentUserDPAPI, generates smaller
  results, and is consistent on all Operating Systems
- you can use this function over a specified variable, to cypher it in place, with try ... finally block to
  protect memory access of the plain data:
    constructor TMyClass.Create;
      ...
      fSecret := CryptDataForCurrentUser('Some Secret Value','appsalt',true);
      ...
    procedure TMyClass.DoSomething;
      var plain: RawByteString;
      begin
        plain := CryptDataForCurrentUser(fSecret,'appsalt',false);
        try
          // here plain = 'Some Secret Value'
        finally
          FillZero(plain); // safely erase uncyphered content from heap
        end;
      end;
    end;

function CryptDataForCurrentUserDPAPI(const Data, AppSecret: RawByteString;Encrypt: boolean): RawByteString;

Protect some data for the current user, using Windows DPAPI
- the application can specify a secret salt text, which should reflect the current execution context,
  to ensure nobody could decrypt the data without knowing this application-specific AppSecret value
- will use CryptProtectData DPAPI function call under Windows
- see https://msdn.microsoft.com/en-us/library/ms995355
- this function is Windows-only, could be slow, and you don't know which algorithm is really used
  on your system, so using CryptDataForCurrentUser() may be a better (and cross-platform)
  alternative
- also note that DPAPI has been closely reverse engineered - see e.g.

procedure FillSystemRandom(Buffer: PByteArray; Len: integer; AllowBlocking: boolean);

Low-level function returning some random binary using standard API
- will call /dev/urandom or /dev/random under POSIX, and CryptGenRandom API on Windows,
  and fallback to SynCommons.FillRandom if the system API failed or for padding if more than 32
  bytes is retrieved from /dev/urandom
- you should not have to call this procedure, but faster and safer TAESPRNG
function Hash128ToDouble(const r: THash128): double;
  Low-level function able to derivate a 0..1 64-bit floating-point from 128-bit of data
  - used e.g. by TAESPRNG.RandomDouble

function Hash128ToExt(const r: THash128): TSynExtended;
  Low-level function able to derivate a 0..1 floating-point from 128-bit of data
  - used e.g. by TAESPRNG.RandomExt

function Hash128ToSingle(const r: THash128): double;
  Low-level function able to derivate a 0..1 32-bit floating-point from 128-bit of data

procedure HashFile(const aFileName: TFileName; aAlgos: THashAlgos); overload;
  Compute the hexadecimal hashe(s) of one file, as external .md5/.sha256/.. files
  - reading the file once in memory, then apply all algorithms on it and generate the text hash files
  in the very same folder

function HashFile(const aFileName: TFileName; aAlgo: THashAlgo): RawUTF8; overload;
  Compute the hexadecimal hash of any (big) file
  - using a temporary buffer of 1MB for the sequential reading

function HashFull(aAlgo: THashAlgo; aBuffer: Pointer; aLen: integer): RawUTF8;
  One-step hash computation of a buffer as lowercase hexadecimal string

procedure HMAC_CRC256C(const key, msg: RawByteString; out result: THash256); overload;
  Compute the HMAC message authentication code using crc256c as hash function
  - HMAC over a non cryptographic hash function like crc256c is known to be safe as MAC, if the
    supplied key comes e.g. from cryptographic HMAC_SHA256
  - performs two crc32c hashes, so SSE 4.2 gives more than 2.2 GB/s on a Core i7

procedure HMAC_CRC256C(key: THash256; const msg: RawByteString; out result: THash256); overload;
  Compute the HMAC message authentication code using crc256c as hash function
  - HMAC over a non cryptographic hash function like crc256c is known to be safe as MAC, if the
    supplied key comes e.g. from cryptographic HMAC_SHA256
  - performs two crc32c hashes, so SSE 4.2 gives more than 2.2 GB/s on a Core i7

procedure HMAC_CRC256C(key: pointer; keylen,msglen: integer; out result: THash256); overload;
  Compute the HMAC message authentication code using crc256c as hash function
  - HMAC over a non cryptographic hash function like crc256c is known to be safe as MAC, if the
    supplied key comes e.g. from cryptographic HMAC_SHA256
  - performs two crc32c hashes, so SSE 4.2 gives more than 2.2 GB/s on a Core i7

function HMAC_CRC32C(const key, msg: RawByteString): cardinal; overload;
  Compute the HMAC message authentication code using crc32c as hash function
  - HMAC over a non cryptographic hash function like crc32c is known to be a safe enough MAC, if
    the supplied key comes e.g. from cryptographic HMAC_SHA256
  - SSE 4.2 will let MAC be computed at 4 GB/s on a Core i7
function HMAC_CRC32C(const key: THash256; const msg: RawByteString): cardinal; overload;
    Compute the HMAC message authentication code using crc32c as hash function
    - HMAC over a non cryptographic hash function like crc32c is known to be a safe enough MAC, if
      the supplied key comes e.g. from cryptographic HMAC_SHA256
    - SSE 4.2 will let MAC be computed at 4 GB/s on a Core i7

function HMAC_CRC32C(key, msg: pointer; keylen, msglen: integer): cardinal; overload;
    Compute the HMAC message authentication code using crc32c as hash function
    - HMAC over a non cryptographic hash function like crc32c is known to be a safe enough MAC, if
      the supplied key comes e.g. from cryptographic HMAC_SHA256
    - SSE 4.2 will let MAC be computed at 4 GB/s on a Core i7

procedure HMAC_SHA1(const key: TSHA1Digest; const msg: RawByteString; out result: TSHA1Digest); overload;
    Compute the HMAC message authentication code using SHA-1 as hash function

procedure HMAC_SHA1(const key,msg: RawByteString; out result: TSHA1Digest); overload;
    Compute the HMAC message authentication code using SHA-1 as hash function

procedure HMAC_SHA1(key, msg: pointer; keylen, msglen: integer; out result: TSHA1Digest); overload;
    Compute the HMAC message authentication code using SHA-1 as hash function

procedure HMAC_SHA256(const key,msg: RawByteString; out result: TSHA256Digest); overload;
    Compute the HMAC message authentication code using SHA-256 as hash function

procedure HMAC_SHA256(key,msg: pointer; keylen,msglen: integer; out result: TSHA256Digest); overload;
    Compute the HMAC message authentication code using SHA-256 as hash function

procedure HMAC_SHA384(const key,msg: RawByteString; out result: TSHA384Digest); overload;
    Compute the HMAC message authentication code using SHA-384 as hash function

procedure HMAC_SHA384(key,msg: pointer; keylen,msglen: integer; out result: TSHA384Digest); overload;
    Compute the HMAC message authentication code using SHA-384 as hash function

procedure HMAC_SHA512(const key,msg: RawByteString; out result: TSHA512Digest); overload;
    Compute the HMAC message authentication code using SHA-512 as hash function

procedure HMAC_SHA512(key,msg: pointer; keylen,msglen: integer; out result: TSHA512Digest); overload;
    Compute the HMAC message authentication code using SHA-512 as hash function

procedure HMAC_SHA384(key,msg: pointer; keylen,msglen: integer; out result: TSHA384Digest); overload;
    Compute the HMAC message authentication code using SHA-384 as hash function

procedure HMAC_SHA512(key,msg: pointer; keylen,msglen: integer; out result: TSHA512Digest); overload;
    Compute the HMAC message authentication code using SHA-512 as hash function
procedure HMAC_SHA512(const key: TSHA512Digest; const msg: RawByteString; out result: TSHA512Digest); overload;

  Compute the HMAC message authentication code using SHA-512 as hash function

function htdigest(const user, realm, pass: RawByteString): RawUTF8;

  Compute the HTDigest for a user and a realm, according to a supplied password
  - apache-compatible: 'agent007:download area:8364d0044ef57b3defcfa141e8f77b65'

function MD5(const s: RawByteString): RawUTF8;

  Direct MD5 hash calculation of some data (string-encoded)
  - result is returned in hexadecimal format

function MD5Buf(const Buffer; Len: Cardinal): TMD5Digest;

  Direct MD5 hash calculation of some data

function MD5DigestToString(const D: TMD5Digest): RawUTF8;

  Compute the hexadecimal representation of a MD5 digest

function MD5SelfTest: boolean;

  Self test of MD5 routines

function MD5StringToDigest(const Source: RawUTF8; out Dest: TMD5Digest): boolean;

  Compute the MD5 digest from its hexadecimal representation
  - returns true on success (i.e. Source has the expected size and characters)
  - just a wrapper around SynCommons.HexToBin()

procedure PBKDF2_HMAC_SHA1(const password, salt: RawByteString; count: Integer; out result: TSHA1Digest);

  Compute the PBKDF2 derivation of a password using HMAC over SHA-1
  - this function expect the resulting key length to match SHA-1 digest size

procedure PBKDF2_HMAC_SHA256(const password, salt: RawByteString; count: Integer; var result: THash256DynArray; const saltdefault: RawByteString=''); overload;

  Compute the PBKDF2 derivation of a password using HMAC over SHA-256, into several 256-bit items, so can be used to return any size of output key
  - this function expect the result array to have the expected output length
  - allows resulting key length to be more than one SHA-256 digest size, e.g. to be used for both Encryption and MAC

procedure PBKDF2_HMAC_SHA256(const password, salt: RawByteString; count: Integer; out result: TSHA256Digest; const saltdefault: RawByteString=''); overload;

  Compute the PBKDF2 derivation of a password using HMAC over SHA-256
  - this function expect the resulting key length to match SHA-256 digest size

procedure PBKDF2_HMAC_SHA384(const password, salt: RawByteString; count: Integer; out result: TSHA384Digest);

  Compute the PBKDF2 derivation of a password using HMAC over SHA-384
  - this function expect the resulting key length to match SHA-384 digest size

procedure PBKDF2_HMAC_SHA512(const password, salt: RawByteString; count: Integer; out result: TSHA512Digest);

  Compute the PBKDF2 derivation of a password using HMAC over SHA-512
  - this function expect the resulting key length to match SHA-512 digest size
procedure PBKDF2_SHA3(algo: TSHA3Algo; const password, salt: RawByteString; count: Integer; result: PByte; resultbytes: integer=0);
  Safe key derivation using iterated SHA-3 hashing
  - you can use SHA3_224, SHA3_256, SHA3_384, SHA3_512 algorithm to fill the result buffer with
    the default sized derivated key of 224, 256, 384 or 512 bits (leaving resultbytes = 0)
  - or you may select SHAKE_128 or SHAKE_256, and specify any custom key size in resultbytes
    (used e.g. by PBKDF2_SHA3_Crypt)

procedure PBKDF2_SHA3_Crypt(algo: TSHA3Algo; const password, salt: RawByteString; count: Integer; var data: RawByteString);
  Encryption/decryption of any data using iterated SHA-3 hashing key derivation
  - specified algo is expected to be SHAKE_128 or SHAKE_256
  - expected the supplied data buffer to be small - for bigger content, consider using TSHA.Cypher
  after 256-bit PBKDF2_SHA3 key derivation

procedure RawMd5Compress(var Hash; Data: pointer);
  Entry point of the raw MD5 transform function - may be used for low-level use

procedure RawSha1Compress(var Hash; Data: pointer);
  Entry point of the raw SHA-1 transform function - may be used for low-level use

procedure RawSha256Compress(var Hash; Data: pointer);
  Entry point of the raw SHA-256 transform function - may be used for low-level use

procedure RawSha512Compress(var Hash; Data: pointer);
  Entry point of the raw SHA-512 transform function - may be used for low-level use

function RC4SelfTest: boolean;
  Self test of RC4 routines

function SHA1(const s: RawByteString): RawUTF8;
  Direct SHA-1 hash calculation of some data (string-encoded)
  - result is returned in hexadecimal format

function SHA1DigestToString(const D: TSHA1Digest): RawUTF8;
  Compute the hexadecimal representation of a SHA-1 digest

function SHA1SelfTest: boolean;
  Self test of SHA-1 routines

function SHA1StringToDigest(const Source: RawUTF8; out Dest: TSHA1Digest): boolean;
  Compute the SHA-1 digest from its hexadecimal representation
  - returns true on success (i.e. Source has the expected size and characters)
  - just a wrapper around SynCommons.HexToBin()

function SHA256(Data: pointer; Len: integer): RawUTF8; overload;
  Direct SHA-256 hash calculation of some binary data
  - result is returned in hexadecimal format

function SHA256(const s: RawByteString): RawUTF8; overload;
  Direct SHA-256 hash calculation of some data (string-encoded)
  - result is returned in hexadecimal format
function SHA256Digest(Data: pointer; Len: integer): TSHA256Digest; overload;

*Direct SHA-256 hash calculation of some binary data*
- result is returned in TSHA256Digest binary format
- since the result would be stored temporarily in the stack, it may be safer to use an explicit TSHA256Digest variable, which would be filled with zeros by a ... finally FillZero()

function SHA256Digest(const Data: RawByteString): TSHA256Digest; overload;

*Direct SHA-256 hash calculation of some binary data*
- result is returned in TSHA256Digest binary format
- since the result would be stored temporarily in the stack, it may be safer to use an explicit TSHA256Digest variable, which would be filled with zeros by a ... finally FillZero()

function SHA256DigestToString(const D: TSHA256Digest): RawUTF8;

*Compute the hexadecimal representation of a SHA-256 digest*

function SHA256SelfTest: boolean;

*Self test of SHA-256 routines*

function SHA256StringToDigest(const Source: RawUTF8; out Dest: TSHA256Digest): boolean;

*Compute the SHA-256 digest from its hexadecimal representation*
- returns true on success (i.e. Source has the expected size and characters)
- just a wrapper around SynCommons.HexToBin()

procedure SHA256Weak(const s: RawByteString; out Digest: TSHA256Digest);

*Direct SHA-256 hash calculation of some data (string-encoded)*
- result is returned in hexadecimal format
- this procedure has a weak password protection: small incoming data is append to some salt, in order to have at least a 256 bytes long hash: such a feature improve security for small passwords, e.g.
- note that this algorithm is proprietary, and less secure (and standard) than the PBKDF2 algorithm, so is there only for backward compatibility of existing code: use PBKDF2_HMAC_SHA256 or similar functions for password derivation

function SHA3(Algo: TSHA3Algo; const s: RawByteString; DigestBits: integer=0): RawUTF8; overload;

*Direct SHA-3 hash calculation of some data (string-encoded)*
- result is returned in hexadecimal format
- default DigestBits=0 will write the default number of bits to Digest output memory buffer, according to the specified TSHA3Algo

function SHA3(Algo: TSHA3Algo; Buffer: pointer; Len: integer; DigestBits: integer=0): RawUTF8; overload;

*Direct SHA-3 hash calculation of some binary buffer*
- result is returned in hexadecimal format
- default DigestBits=0 will write the default number of bits to Digest output memory buffer, according to the specified TSHA3Algo

function SHA384(const s: RawByteString): RawUTF8;

*Direct SHA-384 hash calculation of some data (string-encoded)*
- result is returned in hexadecimal format
function SHA384DigestToString(const D: TSHA384Digest): RawUTF8;
  Compute the hexadecimal representation of a SHA-384 digest

function SHA512(const s: RawByteString): RawUTF8;
  Direct SHA-512 hash calculation of some data (string-encoded)
  - result is returned in hexadecimal format

function SHA512DigestToString(const D: TSHA512Digest): RawUTF8;
  Compute the hexadecimal representation of a SHA-512 digest

procedure XorBlock(p: PIntegerArray; Count, Cod: integer);
  - very fast XOR according to Cod - not Compression or Stream compatible
  - used in AESFull() for KeySize=32

procedure XorBlock16(A,B,C: PCardinalArray); overload;
  Apply the XOR operation to the supplied binary buffers of 16 bytes

procedure XorBlock16(A,B: PCardinalArray); overload;
  Apply the XOR operation to the supplied binary buffers of 16 bytes

procedure XorConst(p: PIntegerArray; Count: integer);
  Fast XOR Cypher changing by Count value
  - Compression compatible, since the XOR value is always the same, the compression rate will not change a lot

procedure XorOffset(P: PByteArray; Index, Count: integer);
  Fast and simple XOR Cypher using Index (=Position in Dest Stream)
  - Compression not compatible with this function: should be applied after compress (e.g. as outStream for TAESWriteStream)
  - Stream compatible (with updated Index)
  - used in AES() and TAESWriteStream

Variables implemented in the SynCrypto unit

AESIVCTR_KEY: TBlock128 = ( $ce5d5e3e, $26506c65, $568e0092, $12cce480);
  128-bit random AES-128 entropy key for TAESAbstract.IVReplayAttackCheck
  - as used internally by AESIVCtrEncryptDecrypt() function
  - you may customize this secret for your own project, but be aware that it will affect all TAESAbstract instances, so should match on all ends

CompressShaAesClass: TAESAbstractClass = TAESCFB;
  The AES-256 encoding class used by CompressShaAes() global function
  - use any of the implementation classes, corresponding to the chaining mode required - TAECBC, TAESCBC, TAESCFB, TAESOFB and TAESTR classes to handle in ECB, CBC, CFB, OFB and CTR mode (including PKCS7-like padding)
  - set to the secure and efficient CFB mode by default

CompressShaAesKey: TSHA256Digest;
  The encryption key used by CompressShaAes() global function
  - the key is global to the whole process
  - use CompressShaAesSetKey() procedure to set this Key from text
CryptProtectDataEntropy: THash256 = (19,8E,8A,F2,0D,99,7B,73,1B,0B,3A,95,7B,73,1B,8B,94,63,C2,C0,78,05,9C,8B,85,87,A1,E3,ED,93,27,18);

Salt for CryptDataForCurrentUser function
- is filled with some random bytes by default, but you may override it for a set of custom processes calling CryptDataForCurrentUser

MainAESPRNG: TAESPRNG;

The shared TAESPRNG instance returned by TAESPRNG.Main class function
- you may override this to a customized instance, e.g. if you expect a specific random generator to be used, like TAESPRNGSystem
- all TAESPRNG.Fill() class functions will use this instance
27.8. SynCurl.pas unit

*Purpose*: Curl library direct access classes
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynCurl unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynWinSock</td>
<td>Low level access to network Sockets for the Win32 platform</td>
<td>1843</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

**Objects implemented in the SynCurl unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECurl</td>
<td>Low-level exception raised during libcurl library access</td>
<td>1200</td>
</tr>
<tr>
<td>TCurlCertInfo</td>
<td>Low-level certificate information for libcurl library API</td>
<td>1200</td>
</tr>
<tr>
<td>TCurlMsgRec</td>
<td>Low-level message information for libcurl library API</td>
<td>1200</td>
</tr>
<tr>
<td>TCurlVersionInfo</td>
<td>Low-level version information for libcurl library</td>
<td>1200</td>
</tr>
<tr>
<td>TCurlWaitFD</td>
<td>Low-level file description event handler for libcurl library API</td>
<td>1200</td>
</tr>
</tbody>
</table>

```pascal
ECurl = class(Exception)
  Low-level exception raised during libcurl library access

TCurlVersionInfo = record
  Low-level version information for libcurl library

TCurlCertInfo = packed record
  Low-level certificate information for libcurl library API

TCurlMsgRec = packed record
  Low-level message information for libcurl library API

TCurlWaitFD = packed record
  Low-level file description event handler for libcurl library API
```

**Types implemented in the SynCurl unit**

```pascal
curl_read_callback = function (buffer: PAnsiChar; size,nitems: integer; instream: pointer): integer; cdecl;
  Low-level read callback function signature for libcurl library API
```
curl_write_callback = function (buffer: PAnsiChar; size,nitems: integer; outstream: pointer): integer; cdecl;

Low-level write callback function signature for libcurl library API

TCurl = type pointer;

Low-level access to the libcurl library instance

TCurlGlobalInit = set of (giNone, giSSL, giWin32, giAll);

Low-level initialization option for libcurl library API
- currently, only giSSL is set, since giWin32 is redundant with WinHTTP


Low-level information enumeration for libcurl library API calls

TCurlMsg = ( cmNone, cmDone);

Low-level message state for libcurl library API

TCurlMulti = type pointer;

Low-level access to the libcurl library instance in "multi" mode

TCurlOption = ( coPort, coTimeout, coInFileSize, coLowSpeedLimit, coLowSpeedTime, coResumeFrom, coCRLF, coSSLVersion, coCondition, coValue, coVerb, coHeader, coNoProgress, coNoBody, coFailOnError, coUpload, coPost, coFPTListOnly, coFTPAppend, coNetRC, coFollowLocation, coTransfer, coPut, coAutoReferer, coProxyPort, coPostFieldSize, coHTTPProxyTunnel, coSSLVerifyPeer, coMaxRedirs, coFileTime, coMaxConnects, coClosePolicy, coFreshConnect, coForbidResue, coConnectTimeout, coHTTPGet, coSSLVerifyHost, coHTTPVersion, coFPTUseEPSP, coSSLEngineDefault, coDNSUseGlobalCache, coNSCacheTimeout, coCookieSession, coBufferSize, coNoSignal, coProxyType, coUnrestrictedAuth, coFPTUseEPRT, coHTTPAuth, coFPCreateMissingDirs, coProxyAuth, coFPTResponseTimeout, coIPResolve, coMaxFileSize, coFPTSSL, coTCPNoDelay, coFPTSSALuth, coignoreContentLength, coFPTStopPassvIp, coFile, coWriteData, coURL, coProxy, coUserPw, coProxyUserPw, coRange, coInFile, coErrorBuffer, coPostFields, coReferer, coFPTPort, coUserAgent, coCookie, coHTTPHeader, coHTTPPost, coSSLCert, coSLLCertPasswd, coQuote, coWriteHeader, coCookieFile, coCustomRequest, coStdErr, coPostQuote, coWriteInfo, coProgressData, coXferInfoData, coInterface, coKRB4Level, coCAInfo, coTelnetOptions, coRandomFile, coEGDSocket, coCookieJar, coSSLCipherList, coSSLCertType, coSSLkeyType, coSSLEngine, coPreQuote, coDebugData, coCapath, coShare, coEncoding, coAcceptEncoding, coPrivate, coHTTP200Aliases, coSSLCtxData, coNetRCFile, coSourceUserPw, coSourcePreQuote, coSourcePostQuote, coIOCTLData, coSourceURL, coSourceQuote, coFPTAccount, coCookieList, coUnixSocketPath, coWriteFunction, coReadFunction, coProgressFunction, coHeaderFunction, coDebugFunction, coSSLContextFunction, coIOCTLFunction, coXferInfoFunction, coInFileSizeLarge, coResumeFromLarge, coMaxFileSizeLarge, coPostFieldLarge );

Low-level options for libcurl library API calls

TCurlResult = ( crOK, crUnsupportedProtocol, crFailedInit, crURLMalfomrat, crURLMalformatUser, crCouldNotResolveProxy, crCouldNotResolveHost, crCouldNotConnect, crFTPWeirdServerReply, crFTPAccessDenied, crFTPUserPasswordIncorrect, crFTPWeirdPassReply, crFTPWeirdUserReply, crFTPWeirdPassVReply, crFTPWeird227Format,
Low-level result codes for libcurl API calls

TCurlSList = type pointer;
Low-level string list type for libcurl library API

TCurlSocket = type TSocket;
Low-level access to one libcurl library socket instance

TCurlVersion = ( cvFirst, cvSecond, cvThird, cvFour, cvLast );
(3) SSL handshake low-level version identifier of the libcurl library

Constants implemented in the SynCurl unit

LIBCURL_DLL = 'libcurl.dll';
Low-level libcurl library file name, depending on the running OS

Functions or procedures implemented in the SynCurl unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurlIsAvailable</td>
<td>Return TRUE if a curl library is available</td>
<td>1202</td>
</tr>
<tr>
<td>CurlWriteRawByteString</td>
<td>Callback used by libcurl to write data: Usage: curl_easy_setopt(fhHandle,coWriteFunction,@CurlWriteRawByteString); curl_easy_setopt(curlHandle,coFile,@curl RespBody); where curlRespBody should be a generic AnsiString/RawByteString, i.e. in practice a SockString or a RawByteString</td>
<td>1203</td>
</tr>
<tr>
<td>LibCurlInitialize</td>
<td>Initialize the libcurl API, accessible via the curl global variable</td>
<td>1203</td>
</tr>
</tbody>
</table>

function CurlIsAvailable: boolean;
Return TRUE if a curl library is available
- will load and initialize it, calling LibCurlInitialize if necessary, catching any exception during the process
function CurlWriteRawByteString(buffer: PAnsiChar; size, nitems: integer; opaque: pointer): integer; cdecl;

Callback used by libcurl to write data; Usage:
curl.easy_setopt(fHandle, coWriteFunction, @CurlWriteRawByteString);
curl.easy_setopt(curlHandle, coFile, @curlRespBody); where curlRespBody should be a generic AnsiString/RawByteString, i.e. in practice a SockString or a RawByteString

procedure LibCurlInitialize(engines: TCurlGlobalInit=[giAll]; const dllname: TFileName= LIBCURL_DLL);

Initialize the libcurl API, accessible via the curl global variable
- do nothing if the library has already been loaded
- will raise ECurl exception on any loading issue

Variables implemented in the SynCurl unit

curl: packed

Low-level late binding functions access to the libcurl library API
- ensure you called LibCurlInitialize or CurlIsAvailable functions to setup this global instance before using any of its internal functions
- see also https://curl.haxx.se/libcurl/c/libcurl-multi.html interface
27.9. SynDB.pas unit

*Purpose:* Abstract database direct access classes
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynDB unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/cache/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

![SynDB class hierarchy diagram]

**Objects implemented in the SynDB unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESQLDBException</td>
<td>Generic Exception type, as used by the SynDB unit</td>
<td>1249</td>
</tr>
<tr>
<td>ESQLDBRemote</td>
<td>Exception raised during remote connection process</td>
<td>1249</td>
</tr>
<tr>
<td>ESQLQueryException</td>
<td>Generic Exception type raised by the TQuery class</td>
<td>1254</td>
</tr>
<tr>
<td>ISQLDBRows</td>
<td>Generic interface to access a SQL query result rows</td>
<td>1210</td>
</tr>
<tr>
<td>ISQLDBStatement</td>
<td>Generic interface to bind to prepared SQL query</td>
<td>1215</td>
</tr>
<tr>
<td>TQuery</td>
<td>Class mapping VCL DB TQuery for direct database process</td>
<td>1257</td>
</tr>
<tr>
<td>TQueryValue</td>
<td>Pseudo-class handling a TQuery bound parameter or column value</td>
<td>1255</td>
</tr>
<tr>
<td>TSQLDBCOLUMNCreate</td>
<td>Used to define how a column to be created</td>
<td>1209</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TSQLDBColumnDefine</td>
<td>Used to define a field/column layout in a table schema</td>
<td>1206</td>
</tr>
<tr>
<td>TSQLDBColumnProperty</td>
<td>Used to define a field/column layout</td>
<td>1208</td>
</tr>
<tr>
<td>TSQLDBConnection</td>
<td>Abstract connection created from TSQLDBConnectionProperties</td>
<td>1230</td>
</tr>
<tr>
<td>TSQLDBConnectionProperties</td>
<td>Abstract class used to set Database-related properties</td>
<td>1219</td>
</tr>
<tr>
<td>TSQLDBConnectionPropertiesThreadSafe</td>
<td>Connection properties which will implement an internal Thread-Safe connection pool</td>
<td>1243</td>
</tr>
<tr>
<td>TSQLDBConnectionThreadSafe</td>
<td>Abstract connection created from TSQLDBConnectionProperties</td>
<td>1243</td>
</tr>
<tr>
<td>TSQLDBDefinitionLimitClause</td>
<td>Defines the LIMIT clause to be inserted for a given SQL syntax</td>
<td>1219</td>
</tr>
<tr>
<td>TSQLDBIndexDefine</td>
<td>Used to describe extended Index definition of a table schema</td>
<td>1206</td>
</tr>
<tr>
<td>TSQLDBLib</td>
<td>Access to a native library</td>
<td>1254</td>
</tr>
<tr>
<td>TSQLDBParam</td>
<td>A structure used to store a standard binding parameter</td>
<td>1244</td>
</tr>
<tr>
<td>TSQLDBProcColumnDefine</td>
<td>Used to define a parameter/column layout in a stored procedure schema</td>
<td>1207</td>
</tr>
<tr>
<td>TSQLDBProxyConnection</td>
<td>Implements an abstract proxy-like virtual connection to a DB engine</td>
<td>1250</td>
</tr>
<tr>
<td>TSQLDBProxyConnectionCommandExecute</td>
<td>Structure to embedd all needed parameters to execute a SQL statement</td>
<td>1249</td>
</tr>
<tr>
<td>TSQLDBProxyConnectionPropertiesAbstract</td>
<td>Implements a proxy-like virtual connection statement to a DB engine</td>
<td>1250</td>
</tr>
<tr>
<td>TSQLDBProxyConnectionProtocol</td>
<td>Server-side implementation of a proxy connection to any SynDB engine</td>
<td>1229</td>
</tr>
<tr>
<td>TSQLDBProxyStatement</td>
<td>Implements a proxy-like virtual connection statement to a DB engine</td>
<td>1252</td>
</tr>
<tr>
<td>TSQLDBProxyStatementAbstract</td>
<td>Implements a proxy-like virtual connection statement to a DB engine</td>
<td>1251</td>
</tr>
<tr>
<td>TSQLDBProxyStatementRandomAccess</td>
<td>Implements a virtual statement with direct data access</td>
<td>1254</td>
</tr>
<tr>
<td>TSQLDBRemoteConnectionPropertiesAbstract</td>
<td>Client-side implementation of a remote connection to any SynDB engine</td>
<td>1253</td>
</tr>
<tr>
<td>TSQLDBRemoteConnectionPropertiesTest</td>
<td>Fake proxy class for testing the remote connection to any SynDB engine</td>
<td>1253</td>
</tr>
<tr>
<td>TSQLDBRemoteConnectionProtocol</td>
<td>Server-side implementation of a remote connection to any SynDB engine</td>
<td>1229</td>
</tr>
<tr>
<td>TSQLDBRowVariantType</td>
<td>A custom variant type used to have direct access to a result row content</td>
<td>1210</td>
</tr>
<tr>
<td>TSQLDBStatement</td>
<td>Generic abstract class to implement a prepared SQL query</td>
<td>1232</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>TSQLDBStatementWithParams</td>
<td>Generic abstract class handling prepared statements with binding</td>
<td>1245</td>
</tr>
<tr>
<td>TSQLDBStatementWithParamsAndColumns</td>
<td>Generic abstract class handling prepared statements with binding and column description</td>
<td>1248</td>
</tr>
</tbody>
</table>

**TSQLDBColumnDefine** = packed record

*Used to define a field/column layout in a table schema*
- for TSQLDBConnectionProperties.SQLCreate to describe the new table
- for TSQLDBConnectionProperties.GetFields to retrieve the table layout

- **ColumnIndexed**: boolean;  
  *Specify if column is indexed*

- **ColumnLength**: PtrInt;  
  *The Column default width (in chars or bytes) of ftUTF8 or ftBlob*  
  - can be set to value <0 for CLOB or BLOB column type, i.e. for a value without any maximal length

- **ColumnName**: RawUTF8;  
  *The Column name*

- **ColumnPrecision**: PtrInt;  
  *The Column data precision*  
  - used e.g. for numerical values

- **ColumnScale**: PtrInt;  
  *The Column data scale*  
  - used e.g. for numerical values  
  - may be -1 if the metadata SQL statement returned NULL

- **ColumnType**: TSQLDBFieldType;  
  *The Column type, as recognized by our SynDB classes*  
  - should not be ftUnknown nor ftNull

- **ColumnTypeNative**: RawUTF8;  
  *The Column type, as retrieved from the database provider*  
  - returned as plain text by GetFields method, to be used e.g. by TSQLDBConnectionProperties.GetFieldDefinitions method  
  - SQLCreate will check for this value to override the default type

**TSQLDBIndexDefine** = packed record

*Used to describe extended Index definition of a table schema*

- **Filter**: RawUTF8;  
  *Expression for the subset of rows included in the filtered index*  
  - only set for MS SQL - not retrieved for other DB types yet
IncludedColumns: RawUTF8;

- Comma separated list of a nonkey column added to the index by using the CREATE INDEX INCLUDE clause
- only set for MS SQL - not retrieved for other DB types yet

IndexName: RawUTF8;

- Name of the index

IsPrimaryKey: boolean;

- If Index is part of a PRIMARY KEY constraint
- only set for MS SQL - not retrieved for other DB types yet

IsUnique: boolean;

- If Index is unique

IsUniqueConstraint: boolean;

- If Index is part of a UNIQUE constraint
- only set for MS SQL - not retrieved for other DB types yet

KeyColumns: RawUTF8;

- Comma separated list of indexed column names, in order of their definition

TypeDesc: RawUTF8;

- Description of the index type
- for MS SQL possible values are: HEAP | CLUSTERED | NONCLUSTERED | XML | SPATIAL
- for Oracle:
  NORMAL | BITMAP | FUNCTION-BASED NORMAL | FUNCTION-BASED BITMAP | DOMAIN

see @http://docs.oracle.com/cd/B19306_01/server.102/b14237/statviews_1069.htm

TSQLDBProcColumnDefine = packed record

- Used to define a parameter/column layout in a stored procedure schema
- for TSQLDBConnectionProperties.GetProcedureParameters to retrieve the stored procedure parameters
- can be extended according to https://msdn.microsoft.com/en-us/library/ms711701(v=vs.85).aspx

ColumnLength: PtrInt;

- The Column default width (in chars or bytes) of ftUTF8 or ftBlob
- can be set to value <0 for CLOB or BLOB column type, i.e. for a value without any maximal length

ColumnName: RawUTF8;

- The Column name

ColumnParamType: TSQLDBParamInOutType;

- Defines the procedure column as a parameter or a result set column

ColumnPrecision: PtrInt;

- The Column data precision
- used e.g. for numerical values
ColumnScale: PtrInt;

*The Column data scale*
- used e.g. for numerical values
- may be -1 if the metadata SQL statement returned NULL

ColumnType: TSQLDBFieldType;

*The Column type, as recognized by our SynDB classes*
- should not be ftUnknown nor ftNull

ColumnTypeNative: RawUTF8;

*The Column type, as retrieved from the database provider*
- used e.g. by TSQLDBConnectionProperties.GetProcedureParameters method

TSQLDBColumnProperty = packed record

*Used to define a field/column layout*
- for TSQLDBConnectionProperties.SQLCreate to describe the table
- for T*Statement.Execute/Column*() methods to map the IRowSet content

ColumnAttr: PtrUInt;

*A general purpose integer value*
- for SQLCreate: default width (in WideChars or Bytes) of ftUTF8 or ftBlob; if set to 0, a CLOB or BLOB column type will be created - note that UTF-8 encoding is expected when calculating the maximum column byte size for the CREATE TABLE statement (e.g. for Oracle 1333=4000/3 is used)
- for TOleDBStatement: the offset of this column in the IRowSet data, starting with a DBSTATUSENUM, the data, then its length (for inlined sftUTF8 and sftBlob only)
- for TSQLDBOracleStatement: contains an offset to this column values inside fRowBuffer[] internal buffer
- for TSQLDBDatasetStatement: maps TField pointer value
- for TSQLDBProxyStatement: contains the column type OID

ColumnDataSize: integer;

*May contain the current column size for not FIXEDLENGTH_SQLDBFIELDTYPE*
- for SynDBODBC: size (in bytes) in corresponding fColData[]
- TSQLDBProxyStatement: the actual maximum column size

ColumnDataState: TSQLDBStatementGetCol;

*May contain the current status of the column value*
- for SynDBODBC: state of the latest SQLGetData() call

ColumnName: RawUTF8;

*The Column name*

ColumnNonNullable: boolean;

*Set if the Column must exists (i.e. should not be null)*

ColumnType: TSQLDBFieldType;

*The Column type, used for storage*
- for SQLCreate: should not be ftUnknown nor ftNull
- for TOleDBStatement: should not be ftUnknown
- for SynDBOracle: never ftUnknown, may be ftNull (for SQT_RSET)
ColumnUnique: Boolean;

*Set if the Column shall have unique value (add the corresponding constraint)*

ColumnValueDBCharSet: Integer;

*Optional character set encoding for ftUTF8 columns*
- for SQLT_STR/SQLT_CLOB (SynDBOracle): equals to the OCI char set

ColumnValueDBForm: Byte;

*Driver-specific encoding information*
- for SynDBOracle: used to store the ftUTF8 column encoding, i.e. for SQLT_CLOB, equals either to SQLCS_NCHAR or SQLCS_IMPLICIT

ColumnValueDBSize: Cardinal;

*Expected column data size*
- for TSQLDBOracleStatement/TOleDBStatement/TODBCStatement: used to store one column size (in bytes)

ColumnValueDBType: Smallint;

*Internal DB column data type*
- for TSQLDBOracleStatement: used to store the DefineByPos() TypeCode, can be SQLT_STR/SQLT_CLOB, SQLT_FLT, SQLT_INT, SQLT_DAT, SQLT_BLOB, SQLT_BIN and SQLT_RSET
- for TSQLDBODBCStatement: used to store the DataType as returned by ODBC.DescribeColW() - use private ODBC_TYPE_TO[ColumnType] to retrieve the marshalled type used during column retrieval
- for TSQLDBFirebirdStatement: used to store XSQLVAR.sqltype
- for TSQLDBDatasetStatement: indicates the TField class type, i.e. 0=TField, 1=TLargeIntField, 2=TWideStringField

ColumnValueInlined: Boolean;

*Set if the Column data is inlined within the main rows buffer*
- for TOleDBStatement: set if column was NOT defined as DBTYPE_BYREF which is the most common case, when column data < 4 KB
- for TSQLDBOracleStatement: FALSE if column is an array of POCIlobLocator (SQLT_CLOB/SQLT_BLOB) or POCIStmt (SQLT_RSET)
- for TSQLDBODBCStatement: FALSE if bigger than 255 WideChar (ftUTF8) or 255 bytes (ftBlob)

TSQLDBColumnCreate = record

*Used to define how a column to be created*

DBType: TSQLDBFieldType;

*The data type*
- here, ftUnknown is used for Int32 values, ftInt64 for Int64 values, as expected by TSQLDBFieldTypeDefinition

Name: RawUTF8;

*The column name*

NonNullable: Boolean;

*If the column should be non null*

PrimaryKey: Boolean;

*If the column is the ID primary key*
Unique: boolean;

    If the column should be unique

Width: cardinal;

    The width, e.g. for VARCHAR() types

TSQLDBRowVariantType = class(TSynInvokeableVariantType)

A custom variant type used to have direct access to a result row content
- use ISQLDBRows.RowData method to retrieve such a Variant

ISQLDBRows = interface(IInterface)

Generic interface to access a SQL query result rows
- not all TSQLDBStatement methods are available, but only those to retrieve data from a statement result: the purpose of this interface is to make easy access to result rows, not provide all available features - therefore you only have access to the Step() and Column*() methods

function ColumnBlob(Col: integer): RawByteString; overload;

    Return a Column as a blob value of the current Row, first Col is 0

function ColumnBlob(const ColName: RawUTF8): RawByteString; overload;

    Return a Column as a blob value of the current Row, from a supplied column name

function ColumnBlobBytes(Col: integer): TBytes; overload;

    Return a Column as a blob value of the current Row, first Col is 0

function ColumnBlobBytes(const ColName: RawUTF8): TBytes; overload;

    Return a Column as a blob value of the current Row, from a supplied column name

function ColumnCount: integer;

    The column/field count of the current Row

function ColumnCurrency(Col: integer): currency; overload;

    Return a Column currency value of the current Row, first Col is 0

function ColumnCurrency(const ColName: RawUTF8): currency; overload;

    Return a Column currency value of the current Row, from a supplied column name

function ColumnCursor(const ColName: RawUTF8): ISQLDBRows; overload;

    Return a special CURSOR Column content as a SynDB result set
- Cursors are not handled internally by mORMot, but some databases (e.g. Oracle) usually use such structures to get data from stored procedures
- such columns are mapped as ftNull internally - so this method is the only one giving access to the data rows

function ColumnCursor(Col: integer): ISQLDBRows; overload;

    Return a special CURSOR Column content as a SynDB result set
- Cursors are not handled internally by mORMot, but some databases (e.g. Oracle) usually use such structures to get data from stored procedures
- such columns are mapped as ftNull internally - so this method is the only one giving access to the data rows
- see also BoundCursor() if you want to access a CURSOR out parameter
function ColumnDateTime(const ColName: RawUTF8): TDateTime; overload;
Return a Column floating point value of the current Row, from a supplied column name

function ColumnDateTime(Col: integer): TDateTime; overload;
Return a Column floating point value of the current Row, first Col is 0

function ColumnDouble(const ColName: RawUTF8): double; overload;
Return a Column floating point value of the current Row, from a supplied column name

function ColumnDouble(Col: integer): double; overload;
Return a Column floating point value of the current Row, first Col is 0

function ColumnIndex(const aColumnName: RawUTF8): integer;
Returns the Column index of a given Column name
* Columns numeration (i.e. Col value) starts with 0
* returns -1 if the Column name is not found (via case insensitive search)

function ColumnInt(const ColName: RawUTF8): Int64; overload;
Return a Column integer value of the current Row, from a supplied column name

function ColumnInt(Col: integer): Int64; overload;
Return a Column integer value of the current Row, first Col is 0

function ColumnName(Col: integer): RawUTF8;
The Column name of the current Row
* Columns numeration (i.e. Col value) starts with 0
* it's up to the implementation to ensure than all column names are unique

function ColumnNull(Col: integer): boolean;
Returns TRUE if the column contains NULL

function ColumnString(const ColName: RawUTF8): string; overload;
Return a Column text value as generic VCL string of the current Row, from a supplied column name

function ColumnString(Col: integer): string; overload;
Return a Column text value as generic VCL string of the current Row, first Col is 0

function ColumnTimestamp(Col: integer): TTimeLog; overload;
Return a column date and time value of the current Row, first Col is 0

function ColumnTimestamp(const ColName: RawUTF8): TTimeLog; overload;
Return a column date and time value of the current Row, from a supplied column name

function ColumnToVariant(Col: integer; var Value: Variant): TSQLDBFieldType;
overload;
Return a Column as a variant, first Col is 0
* this default implementation will call Column*() method above
* a ftUTF8 TEXT content will be mapped into a generic WideString variant for pre-Unico
  version of Delphi, and a generic UnicodeString (=string) since Delphi 2009: you may not loose
  any data during charset conversion
* a ftBlob BLOB content will be mapped into a TBlobData AnsiString variant
function ColumnType(Col: integer; FieldSize: PInteger=nil): TSQLDBFieldType;  
  *The Column type of the current Row*  
- FieldSize can be set to store the size in chars of a ftUTF8 column (0 means BLOB kind of TEXT column)

function ColumnUTF8(const ColName: RawUTF8): RawUTF8; overload;  
  *Return a Column UTF-8 encoded text value of the current Row, from a supplied column name*

function ColumnUTF8(Col: integer): RawUTF8; overload;  
  *Return a Column UTF-8 encoded text value of the current Row, first Col is 0*

function ColumnVariant(Col: integer): Variant; overload;  
  *Return a Column as a variant*  
- a ftUTF8 TEXT content will be mapped into a generic WideString variant for pre-Unicode version of Delphi, and a generic UnicodeString (=string) since Delphi 2009: you may not loose any data during charset conversion
- a ftBlob BLOB content will be mapped into a TBlobData AnsiString variant

function ColumnVariant(const ColName: RawUTF8): Variant; overload;  
  *Return a Column as a variant, from a supplied column name*

function FetchAllAsJSON(Expanded: boolean; ReturnedRowCount: PPtrInt=nil): RawUTF8;  
  *Return all rows content as a JSON string*  
- JSON data is retrieved with UTF-8 encoding
- if Expanded is true, JSON data is an array of objects, for direct use with any Ajax or .NET client: 
  ```json
  [{"col1":val11,"col2":val12},{"col1":val21,...]  
  ```
- if Expanded is false, JSON data is serialized (used in TSQLTableJSON) 
  ```json
  {"FieldCount":1,"Values":["col1","col2",val11,"val12",val21,...] }  
  ```
- BLOB field value is saved as Base64, in the "\uFFF0base64encodedbinary" format and contains true BLOB data
- if ReturnedRowCount points to an integer variable, it will be filled with the number of row data returned (excluding field names)
- similar to corresponding TSQLRequest.Execute method in SynSQLite3 unit

function FetchAllToBinary(Dest: TStream; MaxRowCount: cardinal=0; DataRowPosition: PCardinalDynArray=nil): cardinal;  
  *Append all rows content as binary stream*  
- will save the column types and name, then every data row in optimized binary format (faster and smaller than JSON)
- you can specify a LIMIT for the data extent (default 0 meaning all data)
- generates the format expected by TSQADBProxyStatement
function FetchAllToJSON(JSON: TStream; Expanded: boolean): PtrInt;

Append all rows content as a JSON stream
- JSON data is added to the supplied TStream, with UTF-8 encoding
- if Expanded is true, JSON data is an array of objects, for direct use with any Ajax or .NET client:
  
  \[
  \{
    \"col1\":\"val11\", \"col2\":\"val12\", \"col1\":\"val21\", ...
  \}\n  \]
- if Expanded is false, JSON data is serialized (used in TSQLTableJSON)
  \{
    \"FieldCount\":1, \"Values\": [\"col1\", \"col12\", \"val11\", \"val12\", \"val21\", ...]
  \}\n- BLOB field value is saved as Base64, in the "\\uFF00base64encodedbinary" format and contains true BLOB data
- similar to corresponding TSQLRequest.Execute method in SynSQLite3 unit
- returns the number of row data returned (excluding field names)

function GetColumnVariant(const ColName: RawUTF8): Variant;

Return a Column as a variant, from a supplied column name
- since a property getter can't be an overloaded method, we define one for the Column[] property

function Instance: TSQLDBStatement;

Return the associated statement instance

function RowData: Variant;

Create a TSQLDBRowVariantType able to access any field content via late binding
- i.e. you can use Data.Name to access the 'Name' column of the current row
- this Variant will point to the corresponding TSQLDBStatement instance, so it's not necessary to retrieve its value for each row; but once the associated ISQLDBRows instance is released, you won't be able to access its data - use RowDocVariant instead
- typical use is:
  
  var Row: Variant;
  (...)  
  with MyConnProps.Execute('select * from table where name=?', [aName]) do begin
    Row := RowData;
    while Step do
      writeln(Row.FirstName, Row.BirthDate);
    ReleaseRows;
  end;
function Step(SeekFirst: boolean=false): boolean;

After a prepared statement has been prepared returning a ISQLDBRows interface, this method must be called one or more times to evaluate it:
- you shall call this method before calling any Column*() methods
- return TRUE on success, with data ready to be retrieved by Column*()
- return FALSE if no more row is available (e.g. if the SQL statement is not a SELECT but an UPDATE or INSERT command)
- access the first or next row of data from the SQL Statement result: if SeekFirst is TRUE, will put the cursor on the first row of results, otherwise, it will fetch one row of data, to be called within a loop
- should raise an Exception on any error
- typical use may be:

```pascal
var Customer: Variant;
begin
  with Props.Execute('select * from Sales.Customer where AccountNumber like ?', ['AW000001%'], @Customer) do
    begin
      while Step do // Loop through all matching data rows
        begin
          assert(Copy(Customer.AccountNumber, 1, 8) = 'AW000001');
          ReleaseRows;
          end;
    end;
end;
```

procedure ColumnBlobFromStream(Col: integer; Stream: TStream); overload;
Write a blob Column into the Stream parameter
- expected to be used with 'SELECT .. FOR UPDATE' locking statements

procedure ColumnBlobFromStream(const ColName: RawUTF8; Stream: TStream); overload;
Write a blob Column into the Stream parameter

procedure ColumnBlobToStream(Col: integer; Stream: TStream); overload;
Read a blob Column into the Stream parameter

procedure ColumnBlobToStream(const ColName: RawUTF8; Stream: TStream); overload;
Read a blob Column into the Stream parameter

procedure ColumnToSQLVar(Col: Integer; var Value: TSQLVar; var Temp: RawByteString);
Return a Column as a TSQLVar value, first Col is 0
- the specified Temp variable will be used for temporary storage of svtUTF8/svtBlob values

procedure ReleaseRows;
Release cursor memory and resources once Step loop is finished
- this method call is optional, but is better be used if the ISQLDBRows statement from taken from cache, and returned a lot of content which may still be in client (and server) memory
- will also free all temporary memory used for optional logging

procedure RowDocVariant(out aDocument: variant; aOptions: TDocVariantOptions=JSON_OPTIONS_FAST);
Create a TDocVariant custom variant containing all columns values
- will create a "fast" TDocVariant object instance with all fields
property Column[const ColName: RawUTF8]: Variant read GetColumnVariant;

Return a Column as a variant
- this default property can be used to write simple code like this:

```pascal
procedure WriteFamily(const aName: RawUTF8);
var I: ISQLDBRows;
begin
  I := MyConnProps.Execute('select * from table where name=?',[aName]);
  while I.Step do
    writeln(I['FirstName'], ', ', DateToStr(I['BirthDate']));
  I.ReleaseRows;
end;
```

- of course, using a variant and a column name will be a bit slower than direct access via the Column*() dedicated methods, but resulting code is fast in practice

---

**ISQLDBStatement = interface(ISQLDBRows)**

Generic interface to bind to prepared SQL query
- inherits from ISQLDBRows, so gives access to the result columns data
- not all TSQLDBStatement methods are available, but only those to bind parameters and retrieve data after execution
- reference counting mechanism of this interface will feature statement cache (if available) for NewThreadSafeStatementPrepared() or PrepareInlined()

```pascal
function BoundCursor(Param: Integer): ISQLDBRows;
```

Return a special CURSOR parameter content as a SynDB result set
- this method is not about a column, but a parameter defined with BindCursor() before method execution
- Cursors are not handled internally by mORMot, but some databases (e.g. Oracle) usually use such structures to get data from stored procedures
- this method allow direct access to the data rows after execution

```pascal
function ParamToVariant(Param: Integer; var Value: Variant; CheckIsOutParameter: boolean=true): TSQLDBFieldType;
```

Retrieve the parameter content, after SQL execution
- the leftmost SQL parameter has an index of 1
- to be used e.g. with stored procedures:

```pascal
query := 'BEGIN TEST_PKG.DUMMY(?,?,?,?,?); END;';
stmt := Props.NewThreadSafeStatementPrepared(query, false);
stmt.Bind(1, in1, paramIn);
stmt.BindTextU(2, in2, paramIn);
stmt.BindTextU(3, in3, paramIn);
stmt.BindTextS(4, '', paramOut); // to be retrieved with out1: string
stmt.Bind(5, 0, paramOut); // to be retrieved with out2: integer
stmt.ExecutePrepared;
stmt.ParamToVariant(4, out1, true);
stmt.ParamToVariant(5, out2, true);
```

- the parameter should have been bound with IO=paramOut or IO=paramInOut if CheckIsOutParameter is TRUE

```pascal
function UpdateCount: Integer;
```

Gets a number of updates made by latest executed statement
procedure Bind(Param: Integer; Value: Int64; IO: TSQLDBParamInOutType=paramIn); overload;

  Bind an integer value to a parameter
  - the leftmost SQL parameter has an index of 1

procedure Bind(Param: Integer; const Data: TSQLVar; IO: TSQLDBParamInOutType=paramIn); overload;

  Bind one TSQLVar value
  - the leftmost SQL parameter has an index of 1

procedure Bind(const Params: array of const; IO: TSQLDBParamInOutType=paramIn); overload;

  Bind an array of const values
  - parameters marked as ? should be specified as method parameter in Params[]
  - BLOB parameters can be bound with this method, when set after encoding via BinToBase64WithMagic() call
  - TDateTime parameters can be bound with this method, when encoded via a DateToSQL() or DateTimeToSQL() call

procedure Bind(Param: Integer; ParamType: TSQLDBFieldType; const Value: RawUTF8; ValueAlreadyUnquoted: boolean; IO: TSQLDBParamInOutType=paramIn); overload;

  Bind one RawUTF8 encoded value
  - the leftmost SQL parameter has an index of 1
  - the value should match the BindArray() format, i.e. be stored as in SQL (i.e. number, 'quoted string', 'YYYY-MM-DD hh:mm:ss', null)

procedure Bind(Param: Integer; Value: double; IO: TSQLDBParamInOutType=paramIn); overload;

  Bind a double value to a parameter
  - the leftmost SQL parameter has an index of 1

procedure BindArray(Param: Integer; const Values: array of double); overload;

  Bind an array of double values to a parameter
  - the leftmost SQL parameter has an index of 1
  - this default implementation will raise an exception if the engine does not support array binding

procedure BindArray(Param: Integer; const Values: array of Int64); overload;

  Bind an array of integer values to a parameter
  - the leftmost SQL parameter has an index of 1
  - this default implementation will raise an exception if the engine does not support array binding

procedure BindArray(Param: Integer; const Values: array of RawUTF8); overload;

  Bind an array of RawUTF8 values to a parameter
  - the leftmost SQL parameter has an index of 1
  - values are stored as in SQL (i.e. 'quoted string')
  - this default implementation will raise an exception if the engine does not support array binding

procedure BindArray(Param: Integer; ParamType: TSQLDBFieldType; const Values: TRawUTF8DynArray; ValuesCount: integer); overload;

  Bind an array of values to a parameter
  - the leftmost SQL parameter has an index of 1
  - values are stored as in SQL (i.e. number, 'quoted string', 'YYYY-MM-DD hh:mm:ss', null)
  - this default implementation will raise an exception if the engine does not support array binding
procedure BindArrayCurrency(Param: Integer; const Values: array of currency);

Bind an array of currency values to a parameter
- the leftmost SQL parameter has an index of 1
- this default implementation will raise an exception if the engine does not support array binding

procedure BindArrayDateTime(Param: Integer; const Values: array of TDateTime);

Bind an array of TDateTime values to a parameter
- the leftmost SQL parameter has an index of 1
- values are stored as in SQL (i.e. 'YYYY-MM-DD hh:mm:ss')
- this default implementation will raise an exception if the engine does not support array binding

procedure BindBlob(Param: Integer; Data: pointer; Size: integer; IO: TSQLDBParamInOutType=paramIn); overload;

Bind a Blob buffer to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindBlob(Param: Integer; const Data: RawByteString; IO: TSQLDBParamInOutType=paramIn); overload;

Bind a Blob buffer to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindCurrency(Param: Integer; Value: currency; IO: TSQLDBParamInOutType=paramIn); overload;

Bind a currency value to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindCursor(Param: integer);

Bind a special CURSOR parameter to be returned as a SynDB result set
- Cursors are not handled internally by mORMot, but some databases (e.g. Oracle) usually use such structures to get data from stored procedures
- such parameters are mapped as ftUnknown
- use BoundCursor() method to retrieve the corresponding ISQLDBRows after execution of the statement

procedure BindDateTime(Param: Integer; Value: TDateTime; IO: TSQLDBParamInOutType=paramIn); overload;

Bind a TDateTime value to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindFromRows(const Fields: TSQLDBFieldTypeDynArray; Rows: TSQLDBStatement);

Bind an array of fields from an existing SQL statement
- can be used e.g. after ColumnsToSQLInsert() method call for fast data conversion between tables

procedure BindNull(Param: Integer; IO: TSQLDBParamInOutType=paramIn; BoundType: TSQLDBFieldType=ftNull);

Bind a NULL value to a parameter
- the leftmost SQL parameter has an index of 1
- some providers (e.g. OleDB during MULTI INSERT statements) expect the proper column type to be set in BoundType, even for NULL values
procedure BindTextP(Param: Integer; Value: PUTF8Char; IO: TSQLDBParamInOutType=paramIn); overload;

Bind a UTF-8 encoded buffer text (#0 ended) to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindTextS(Param: Integer; const Value: string; IO: TSQLDBParamInOutType=paramIn); overload;

Bind a UTF-8 encoded string to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindTextU(Param: Integer; const Value: RawUTF8; IO: TSQLDBParamInOutType=paramIn); overload;

Bind a UTF-8 encoded string to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindTextW(Param: Integer; const Value: WideString; IO: TSQLDBParamInOutType=paramIn); overload;

Bind a UTF-8 encoded string to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindVariant(Param: Integer; const Data: Variant; DataIsBlob: boolean; IO: TSQLDBParamInOutType=paramIn);

Bind a Variant value to a parameter
- the leftmost SQL parameter has an index of 1
- will call all virtual Bind*() methods from the Data type
- if DataIsBlob is TRUE, will call BindBlob(RawByteString(Data)) instead of BindTextW(WideString(Variant)) - used e.g. by TQuery.AsBlob/AsBytes

procedure ExecutePrepared;

Execute a prepared SQL statement
- parameters marked as ? should have been already bound with Bind*() functions
- should raise an Exception on any error
- after execution, you can access any returned data via ISQLDBRows methods

procedure ExecutePreparedAndFetchAllAsJSON(Expanded: boolean; out JSON: RawUTF8);

Execute a prepared SQL statement and return all rows content as a JSON string
- JSON data is retrieved with UTF-8 encoding
- if Expanded is true, JSON data is an array of objects, for direct use with any Ajax or .NET client:
  
  ```json
  [ { "col1":val11,"col2":val12" }, { "col1":val12,... } ]
  ```
- if Expanded is false, JSON data is serialized (used in TSQLiteTableJSON)
  
  ```json
  { "FieldCount":1,"Values":["col1","col2",val11,"val12",val21,...] }  
  ```
- BLOB field value is saved as Base64, in the "\uffff0base64encodedbinary" format and contains true BLOB data

property ForceBlobAsNull: boolean read GetForceBlobAsNull write SetForceBlobAsNull;

If set, any BLOB field won't be retrieved, and forced to be null
- this may be used to speed up fetching the results for SQL requests with * statements

property ForceDateWithMS: boolean read GetForceDateWithMS write SetForceDateWithMS;

If set, any ftDate field will contain the milliseconds information when serialized into ISO-8601 text
- this setting is private to each statement, since may vary depending on data definition (e.g. ORM TDateTime/TDateTimeMS)
### TSQLDBDefinitionLimitClause = record

*Defines the LIMIT clause to be inserted for a given SQL syntax*
- used by TSQLDBDefinitionLimitClause and SQLLimitClause() method

### TSQLDBConnectionProperties = class(TObject)

*Abstract class used to set Database-related properties*
- handle e.g. the Database server location and connection parameters (like UserID and password)
- should also provide some Database-specific generic SQL statement creation (e.g. how to create a Table), to be used e.g. by the mORMot layer
- this class level will handle a single "main connection" - you may inherit from TSQLDBConnectionThreadSafe to maintain one connection per thread

**constructor** Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8);
**virtual**;

*Initialize the properties*
- children may optionally handle the fact that no UserID or Password is supplied here, by displaying a corresponding Dialog box

**destructor** Destroy; **override**;

*Release related memory, and close MainConnection*

**class function** ClassFrom(aDefinition: TSynConnectionDefinition): TSQLDBConnectionPropertiesClass;

*Retrieve the registered class from the aDefinition.Kind string*

**function** ColumnTypeNativeToDB(const aNativeType: RawUTF8; aScale: integer): TSQLDBFieldType; **virtual**;

*Convert a textual column data type, as retrieved e.g. from SQLGetField, into our internal primitive types*
- default implementation will always return ftUTF8

**class function** CreateFrom(aDefinition: TSynConnectionDefinition): TSQLDBConnectionProperties; **virtual**;

*Create a new TSQLDBConnectionProperties instance from the stored values*

**class function** CreateFromFile(const aJSONFile: TFileName; aKey: cardinal=0): TSQLDBConnectionProperties;

*Create a new TSQLDBConnectionProperties instance from a JSON file*
- as previously serialized with TSQLDBConnectionProperties.DefinitionToFile
- you can specify a custom Key, if the default is not safe enough for you

**class function** CreateFromJSON(const aJSONDefinition: RawUTF8; aKey: cardinal=0): TSQLDBConnectionProperties; **virtual**;

*Create a new TSQLDBConnectionProperties instance from a JSON content*
- as previously serialized with TSQLDBConnectionProperties.DefinitionToJSON
- you can specify a custom Key, if the default is not safe enough for you

**function** DefinitionToJSON(Key: cardinal=0): RawUTF8; **virtual**;

*Save the properties into a JSON file*
- you could use TSQLDBConnectionPropertiesDescription.CreateFromJSON() later on to instantiate the proper TSQLDBConnectionProperties class
- you can specify a custom Key, if the default is not enough for you
class function EngineName: RawUTF8;
  // Return the database engine name, as computed from the class name
  // 'TSQLDBConnectionProperties' will be trimmed left side of the class name
function Execute(const aSQL: RawUTF8; const Params: array of const; RowsVariant: PVariant=nil; ForceBlobAsNull: boolean=false): ISQLDBRows;
  // Execute a SQL query, returning a statement interface instance to retrieve the result rows
  // corresponding to the supplied SELECT statement
  // - will call NewThreadSafeStatement method to retrieve a thread-safe statement instance, then
  //    run the corresponding Execute() method
  // - raise an exception on error
  // - returns an ISQLDBRows to access any resulting rows (if ExpectResults is TRUE), and provide
  // basic garbage collection, as such:
  // procedure WriteFamily(const aName: RawUTF8);
  // var I: ISQLDBRows;
  // begin
  //   I := MyConnProps.Execute('select * from table where name=?',[aName]);
  //   while I.Step do
  //     writeln(I['FirstName'], ', DateToStr(I['BirthDate']));
  //   I.ReleaseRows;
  // end;
  // - if RowsVariant is set, you can use it to row column access via late binding, as such:
  // procedure WriteFamily(const aName: RawUTF8);
  // var R: Variant;
  // begin
  //   with MyConnProps.Execute('select * from table where name=?',[aName],[@R]) do begin
  //     while Step do
  //       writeln(R.FirstName, ', DateToStr(R.BirthDate));
  //   ReleaseRows;
  // end;
  // - you can any BLOB field to be returned as null with the ForceBlobAsNull optional parameter
function ExecuteInlined(const SQLFormat: RawUTF8; const Args: array of const; ExpectResults: Boolean): ISQLDBRows; overload;
  // Create, prepare, bound inlined parameters and execute a thread-safe statement
  // - overloaded method using FormatUTF8() and inlined parameters
function ExecuteInlined(const aSQL: RawUTF8; ExpectResults: Boolean): ISQLDBRows; overload;
  // Create, prepare, bound inlined parameters and execute a thread-safe statement
  // - this implementation will call the NewThreadSafeStatement virtual method, then bound inlined
  //   parameters as :(1234): and call its Execute method
  // - raise an exception on error
function ExecuteNoResult(const aSQL: RawUTF8; const Params: array of const): integer;
  // Execute a SQL query, without returning any rows
  // - can be used to launch INSERT, DELETE or UPDATE statement, e.g.
  // - will call NewThreadSafeStatement method to retrieve a thread-safe statement instance, then
  //   run the corresponding Execute() method
  // - return the number of modified rows, i.e. the ISQLDBStatement.UpdateCount value (or 0 if the
  //   DB driver does not supply this value)
class function GetFieldDefinition(const Column: TSQLDBColumnDefine): RawUTF8;

Get one field/column definition as text
- return column type as 'Name [Type Length Precision Scale]'.

class function GetFieldORMDefinition(const Column: TSQLDBColumnDefine): RawUTF8;

Get one field/column definition as text, targeting a TSQLRecord published property
- return e.g. property type information as:
  'Name: RawUTF8 read fName write fName index 20;';

function GetForeignKey(const aTableName, aColumnName: RawUTF8): RawUTF8;

Retrieve a foreign key for a specified table and column
- first time it is called, it will retrieve all foreign keys from the remote database using virtual protected GetForeignKeys method into the protected fForeignKeys list: this may be slow, depending on the database access (more than 10 seconds waiting is possible)
- any further call will use this internal list, so response will be immediate
- the whole foreign key list is shared by all connections

function IsCachable(P: PUTF8Char): boolean; virtual;
Determine if the SQL statement can be cached
- used by TSQLDBConnection.NewStatementPrepared() for handling cache

function IsSQLKeyword(aWord: RawUTF8): boolean; overload;
Check if the supplied text word is not a keyword for the current database engine
- just a wrapper around the overloaded class function

class function IsSQLKeyword(aDB: TSQLDBDefinition; aWord: RawUTF8): boolean; overload; virtual;
Check if the supplied text word is not a keyword for a given database engine

function NewConnection: TSQLDBConnection; virtual;
Create a new connection
- call this method if the shared MainConnection is not enough (e.g. for multi-thread access)
- the caller is responsible of freeing this instance

function NewThreadSafeStatement: TSQLDBStatement;
Create a new thread-safe statement
- this method will call ThreadSafeConnection.NewStatement

function NewThreadSafeStatementPrepared(const SQLFormat: RawUTF8; const Args: array of const; ExpectResults: Boolean; RaiseExceptionOnError: Boolean=false): ISQLDBStatement; overload;
Create a new thread-safe statement from an internal cache (if any)
- this method will call the overloaded NewThreadSafeStatementPrepared method
- here Args[] array does not refer to bound parameters, but to values to be changed within SQLFormat in place of '%' characters (this method will call FormatUTF8() internally); parameters will be bound directly on the returned TSQLDBStatement instance
- this method should return a prepared statement instance on success
- on error, returns nil and you can check Connection.LastErrorMessage / Connection.LastErrorException to retrieve corresponding error information (if RaiseExceptionOnError is left to default FALSE value, otherwise, it will raise an exception)
function NewThreadSafeStatementPrepared(const aSQL: RawUTF8; ExpectResults: Boolean; RaiseExceptionOnError: Boolean=false): ISQLDBStatement; overload;

Create a new thread-safe statement from an internal cache (if any)
- will call ThreadSafeConnection.NewStatementPrepared
- this method should return a prepared statement instance on success
- on error, returns nil and you can check Connection.LastErrorMessage / Connection.LastErrorException to retrieve corresponding error information (if RaiseExceptionOnError is left to default FALSE value, otherwise, it will raise an exception)

function PrepareInlined(const SQLFormat: RawUTF8; const Args: array of const; ExpectResults: Boolean): ISQLDBStatement; overload;

Create, prepare and bound inlined parameters to a thread-safe statement
- overloaded method using FormatUTF8() and inlined parameters
- consider using ExecuteInlined() for direct execution

function PrepareInlined(const aSQL: RawUTF8; ExpectResults: Boolean): ISQLDBStatement; overload;

Create, prepare and bound inlined parameters to a thread-safe statement
- this implementation will call the NewThreadSafeStatement virtual method, then bind inlined parameters as :{(1234)}: and return the resulting statement
- raise an exception on error
- consider using ExecuteInlined() for direct execution

function SharedTransaction(SessionID: cardinal; action: TSQLDBSharedTransactionAction): TSQLDBConnection; virtual;

Handle a transaction process common to all associated connections
- could be used to share a single transaction among several connections, or to run nested transactions even on DB engines which do not allow them
- will use a simple reference counting mechanism to allow nested transactions, identified by a session identifier
- will fail if the same connection is not used for the whole process, which would induce a potentially incorrect behavior
- returns the connection corresponding to the session, nil on error

function SQLAddColumn(const aTableName: RawUTF8; const aField: TSQLDBColumnCreate): RawUTF8; virtual;

Returns the SQL statement used to add a column to a Table
- should return the SQL "ALTER TABLE" statement needed to add a column to an existing table
- this default implementation will use internal TSQLCreateField and fSQLCreateFieldMax protected values, which contains by default the ANSI SQL Data Types and maximum 1000 inlined WideChars: inherited classes may change the default fSQLCreateField* content or override this method

function SQLAddIndex(const aTableName: RawUTF8; const aFieldNames: array of RawUTF8; aUnique: boolean; aDescending: boolean=false; const aIndexName: RawUTF8=''): RawUTF8; virtual;

Returns the SQL statement used to add an index to a Table
- should return the SQL "CREATE INDEX" statement needed to add an index to the specified column names of an existing table
- index will expect UNIQUE values in the specified columns, if Unique parameter is set to true
- this default implementation will return the standard SQL statement, i.e. 'CREATE [UNIQUE] INDEX index_name ON table_name (column_name[s])'
Synopsis ORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

function SQLCreate(const aTableName: RawUTF8; const aFields: TSQLDBColumnCreateDynArray; aAddID: boolean): RawUTF8; virtual;

- should return the SQL "CREATE" statement needed to create a table with the specified field/column names and types
- if aAddID is TRUE, "ID Int64 PRIMARY KEY" column is added as first, and will expect the ORM to create a unique RowID value sent at INSERT (could use "select max(ID) from table" to retrieve the last value) - note that 'ID' is used instead of 'RowID' since it fails on Oracle e.g.
- this default implementation will use internal fSQLCreateField and fSQLCreateFieldMax protected values, which contains by default the ANSI SQL Data Types and maximum 1000 inlined WideChars: inherited classes may change the default fSQLCreateField* content or override this method

function SQLCreateDatabase(const aDatabaseName: RawUTF8; aDefaultPageSize: integer=0): RawUTF8;

- this default implementation will only handle dFirebird by now

function SQLDateToIso8601Quoted(DateTime: TDateTime): RawUTF8;

- e.g. SQLite3, DB2 and PostgreSQL will use non-standard '\'' instead of '"'

function SQLFullTableName(const aTableName: RawUTF8): RawUTF8;

- will use ForcedSchemaName property (if applying), or return aTableName
- you can override this method to force the expected format

function SQLIso8601ToDate(const Iso8601: RawUTF8): RawUTF8;

- 'YYYY-MM-DDTHH:MM:SS' (as expected by Microsoft SQL server e.g.)
- returns "TO_DATE('....','YYYY-MM-DD HH24:MI:SS')" for Oracle

function SQLLimitClause(AStmt: TSynTableStatement): TSQLDBDefinitionLimitClause;

- this default implementation will return the quoted ISO-8601 value, i.e. 'YYYY-MM-DDTHH:MM:SS'
- returns to_date('....','YYYY-MM-DD HH24:MI:SS') for Oracle

function SQLSelectAll(const aTableName: RawUTF8; const aFields: TSQLDBColumnDefineDynArray; aExcludeTypes: TSQLDBFieldTypes): RawUTF8; virtual;

- should return the SQL "SELECT ... FROM ..." statement to retrieve the specified column names of an existing table
- by default, all columns specified in aFields[] will be available: it will return "SELECT * FROM TableName"
- but if you specify a value in aExcludeTypes, it will compute the matching column names to ignore those kind of content (e.g. [stBlob] to save time and space)
function SQLTableName(const aTableName: RawUTF8): RawUTF8; virtual;

Return a SQL table name with quotes if necessary
- can be used e.g. with SELECT statements
- you can override this method to force the expected format

function ThreadSafeConnection: TSQLDBConnection; virtual;

Get a thread-safe connection
- this default implementation will return the MainConnection shared instance, so the provider should be thread-safe by itself
- TSQLDBConnectionPropertiesThreadSafe will implement a per-thread connection pool, via an internal TSQLDBConnection pool, per thread if necessary (e.g. for OleDB, which expect one TOleDBConnection instance per thread)

procedure ClearConnectionPool; virtual;

Release all existing connections
- can be called e.g. after a DB connection problem, to purge the connection pool, and allow automatic reconnection
- is called automatically if ConnectionTimeOutMinutes property is set
- warning: no connection shall still be used on the background (e.g. in multi-threaded applications), or some unexpected border effects may occur

procedure DefinitionTo(Definition: TSynConnectionDefinition); virtual;

Save the properties into a persistent storage object
- you can use TSQLDBConnectionPropertiesDescription.CreateFrom() later on to instantiate the proper TSQLDBConnectionProperties class
- current Definition.Key value will be used for the password encryption

procedure DefinitionToFile(const aJSONFile: TFileName; Key: cardinal=0);

Save the properties into a JSON file
- you could use TSQLDBConnectionPropertiesDescription.CreateFromFile() later on to instantiate the proper TSQLDBConnectionProperties class
- you can specify a custom Key, if the default is not enough for you

procedure GetFieldDefinitions(const aTableName: RawUTF8; out Fields: TRawUTF8DynArray; WithForeignKeys: boolean);

Get all field/column definition for a specified Table as text
- call the GetFields method and retrieve the column field name and type as 'Name [Type Length Precision Scale]'
- if WithForeignKeys is set, will add external foreign keys as '% tablename'

procedure GetFields(const aTableName: RawUTF8; out Fields: TSQLDBColumnDefineDynArray); virtual;

Retrieve the column/field layout of a specified table
- this default implementation will use protected SQLGetField virtual method to retrieve the field names and properties
- used e.g. by GetFieldDefinitions
- will call ColumnTypeNativeToDB protected virtual method to guess the each mORMot TSQLDBFieldType
procedure GetIndexes(const aTableName: RawUTF8; out Indexes: TSQLDBIndexDefineDynArray); virtual;

Retrieve the advanced indexed information of a specified Table
- this default implementation will use protected SQLGetIndex virtual method to retrieve the index names and properties
- currently only MS SQL and Oracle are supported

procedure GetProcedureNames(out Procedures: TRawUTF8DynArray); virtual;

Retrieve a list of stored procedure names from current connection

procedure GetProcedureParameters(const aProcName: RawUTF8; out Parameters: TSQLDBProcColumnDefineDynArray); virtual;

Retrieve procedure input/output parameter information
- aProcName: stored procedure name to retrieve parameter information.
- Parameters: parameter list info (name, datatype, direction, default)

procedure GetTableNames(out Tables: TRawUTF8DynArray); virtual;

Get all table names
- this default implementation will use protected SQLGetTableNames virtual method to retrieve the table names

procedure GetViewNames(out Views: TRawUTF8DynArray); virtual;

Get all view names
- this default implementation will use protected SQLGetViewNames virtual method to retrieve the view names

procedure SQLSplitProcedureName(const aProcName: RawUTF8; out Owner, Package, ProcName: RawUTF8); virtual;

Split a procedure name to its OWNER.PACKAGE.PROCEDURE full name (if applying)
- will use ForcedSchemaName property (if applying), or the OWNER. already available within the supplied table name

procedure SQLSplitTableName(const aTableName: RawUTF8; out Owner, Table: RawUTF8); virtual;

Split a table name to its OWNER.TABLE full name (if applying)
- will use ForcedSchemaName property (if applying), or the OWNER. already available within the supplied table name

property BatchMaxSentAtOnce: integer read fBatchMaxSentAtOnce write fBatchMaxSentAtOnce;

The maximum number of rows to be transmitted at once for batch sending
- e.g. Oracle handles array DML operation with iters <= 32767 at best
- if OnBatchInsert points to MultipleValuesInsert(), this value is ignored, and the maximum number of parameters is guessed per DBMS type

property BatchSendingAbilities: TSQLDBStatementCRUDs read fBatchSendingAbilities;

The abilities of the database for batch sending
- e.g. Oracle will handle array DML binds, or MS SQL bulk insert
property ConnectionTimeOutMinutes: cardinal read GetConnectionTimeOutMinutes write SetConnectionTimeOutMinutes;

Specify a maximum period of inactivity after which all connections will be flushed and recreated, to avoid potential broken connections issues.
- In practice, recreating the connections after a while is safe and won’t slow done the process - on the contrary, it may help reducing the consumed resources, and stabilize long running n-Tier servers.
- ThreadSafeConnection method will check for the last activity on this TSQLDBConnectionProperties instance, then call ClearConnectionPool to release all active connections if the idle time elapsed was too long.
- Warning: no connection shall still be used on the background (e.g. in multi-threaded applications), or some unexpected issues may occur - for instance, ensure that your mORMot ORM server runs all its statements in blocking mode for both read and write:

```pascal
aServer.AcquireExecutionMode[execORMGet] := am***;
aServer.AcquireExecutionMode[execORMWrite] := am***;
```

Here, safe blocking am*** modes are any mode but amUnlocked, i.e. either amLocked, amBackgroundThread or amMainThread.

property DatabaseName: RawUTF8 read fDatabaseName;

The associated database name, as specified at creation.
- Not published, for security reasons (may be serialized otherwise).
- DatabaseNameSafe will be published, and delete any matching PasswordValue in DatabaseName.

property DatabaseNameSafe: RawUTF8 read GetDatabaseNameSafe;

The associated database name, safely trimmed from the password.
- Would replace any matching Password value content from DatabaseName by '***' for security reasons, e.g. before serialization.

property DateTimeFirstChar: AnsiChar read fDateTimeFirstChar write fDateTimeFirstChar;

Customize the ISO-8601 text format expected by the database provider.
- Is 'T' by default, as expected by the ISO-8601 standard.
- Will be changed e.g. for PostgreSQL, which expects '' instead.
- As used by SQLDateToIso8601Quoted() and BindArray().

property DBMS: TSQLDBDefinition read GetDBMS;

The remote DBMS type, as stated by the inheriting class itself, or retrieved at connecton time (e.g. for ODBC).

property DBMSEngineName: RawUTF8 read GetDBMSName;

The remote DBMS type name, retrieved as text from the DBMS property.

property Engine: RawUTF8 read fEngineName;

Return the database engine name, as computed from the class name.
- 'TSQLDBConnectionProperties' will be trimmed left side of the class name.
property ExecuteWhenConnected: TRawUTF8DynArray read fExecuteWhenConnected write fExecuteWhenConnected;

SQL statements what will be executed for each new connection usage scenarios examples:
- Oracle: force case-insensitive like
  ['ALTER SESSION SET NLS_COMP=LINGUISTIC', 'ALTER SESSION SET NLS_SORT=BINARY_CI']
- Postgres: disable notices and warnings
  ['SET client_min_messages to ERROR']
- SQLite3: turn foreign keys ON
  ['PRAGMA foreign_keys = ON']

property FilterTableViewSchemaName: boolean read fFilterTableViewSchemaName write fFilterTableViewSchemaName;

If GetTableNames/GetViewNames should only return the table names starting with 'ForcedSchemaName.' prefix

property ForcedSchemaName: RawUTF8 read fForcedSchemaName write fForcedSchemaName;
An optional Schema name to be used for SQLGetField() instead of UserID
- by default, UserID will be used as schema name, if none is specified (i.e. if table name is not set as SCHEMA.TABLE)
- depending on the DBMS identified, the class may also set automatically the default 'dbo' for MS SQL or 'public' for PostgreSQL
- you can set a custom schema to be used instead

property ForeignKeysData: RawByteString read GetForeignKeysData write SetForeignKeysData;
Can be used to store the fForeignKeys[] data in an external BLOB
- since GetForeignKeys can be (somewhat) slow, could save a lot of time

property LoggedSQLMaxSize: integer read fLoggedSQLMaxSize write fLoggedSQLMaxSize;
The maximum size, in bytes, of logged SQL statements
- setting 0 will log statement and parameters with no size limit
- setting -1 will log statement without any parameter value (just ?)
- setting any value >0 will log statement and parameters up to the number of bytes (default set to 2048 to log up to 2KB per statement)

property LogSQLStatementOnException: boolean read fLogSQLStatementOnException write fLogSQLStatementOnException;
Allow to log the SQL statement when any low-level ESQLDBException is raised

property MainConnection: TSQLDBConnection read GetMainConnection;
Return a shared connection, corresponding to the given database
- call the ThreadSafeConnect method instead e.g. for multi-thread access, or
NewThreadSafeStatement for direct retrieval of a new statement

property OnBatchInsert: TOnBatchInsert read fOnBatchInsert write fOnBatchInsert;
You can define a callback method able to handle multiple INSERT
- may execute e.g. INSERT with multiple VALUES (like MySQL, MSSQL, NexusDB, PostgreSQL or SQLite3), as defined by MultipleValuesInsert() callback
   This event handler will be called during all process
   - can be used e.g. to change the desktop cursor, or be notified on
     connection/disconnection/reconnection
   - you can override this property directly in the TSQLDBConnection

property OnStatementInfo: TOnSQLDBInfo read fOnStatementInfo write fOnStatementInfo;
   This event handler will be called when statements trigger some low-level information

property PassWord: RawUTF8 read fPassWord;
   The associated User Password, as specified at creation
   - not published, for security reasons (may be serialized otherwise)

property ReconnectAfterConnectionError: boolean read fReconnectAfterConnectionError write fReconnectAfterConnectionError;
   Intercept connection errors at statement preparation and try to reconnect
   - i.e. detect TSQLDBConnection.LastErrorWasAboutConnection in
     TSQLDBConnection.NewStatementPrepared
   - warning: no connection shall still be used on the background (e.g. in multi-threaded
     applications), or some unexpected issues may occur - see AcquireExecutionMode[]
     recommendations in ConnectionTimeOutMinutes

property RollbackOnDisconnect: Boolean read fRollbackOnDisconnect write fRollbackOnDisconnect;
   Defines if TSQLDBConnection.Disconnect shall Rollback any pending transaction
   - some engines executes a COMMIT when the client is disconnected, others do raise an
     exception: this parameter ensures that any pending transaction is roll-backed before
     disconnection
   - is set to TRUE by default

property ServerName: RawUTF8 read fServerName;
   The associated server name, as specified at creation

property StatementCacheReplicates: integer read fStatementCacheReplicates write fStatementCacheReplicates;
   If UseCache is true, how many statement replicates can be generated if the cached
   ISQLDBStatement is already used
   - such replication is normally not needed in a per-thread connection, unless ISQLDBStatement
     are not released as soon as possible
   - above this limit, no cache will be made, and a dedicated single-time statement will be prepared
   - default is 0 to cache statements once - but you may try to increase this value if you run
     identical SQL with long-standing ISQLDBStatement; or you can set -1 if you don't want the
     warning log to appear

property StoreVoidStringAsNull: Boolean read fStoreVoidStringAsNull write fStoreVoidStringAsNull;
   Defines if " string values are to be stored as SQL null
   - by default, " will be stored as"
   - but some DB engines (e.g. Jet or MS SQL) does not allow by default to store " values, but
     expect NULL to be stored instead
property UseCache: boolean read fUseCache write fUseCache;

TRUE if an internal cache of SQL statement should be used
- cache will be accessed for NewStatementPrepared() method only, by returning
  ISQLDBStatement interface instances
- default value is TRUE for faster process (e.g. TTestSQLite3ExternalDB regression tests will be
two times faster with statement caching)
- will cache only statements containing ? parameters or a SELECT with no WHERE clause within

property UserID: RawUTF8 read fUserID;

  The associated User Identifier, as specified at creation

property VariantAsStringAsWideString: boolean read fVariantWideString write fVariantWideString;

  Set to true to force all variant conversion to WideString instead of the default faster AnsiString,
  for pre-Unicode version of Delphi
- by default, the conversion to Variant will create an AnsiString kind of variant: for pre-Unidecode
  Delphi, avoiding WideString/OleStr content will speed up the process a lot, if you are sure that
  the current charset matches the expected one (which is very likely)
- set this property to TRUE so that the conversion to Variant will create a WideString kind of
  variant, to avoid any character data loss: the access to the property will be slower, but you won’t
  have any potential data loss
- starting with Delphi 2009, the TEXT content will be stored as an UnicodeString in the variant, so
  this property is not necessary
- the Variant conversion is mostly used for the TQuery wrapper, or for the ISQLDBRows.Column[]
  property or ISQLDBRows.ColumnVariant() method; this won’t affect other Column*() methods,
  or JSON production

TSQLDBProxyConnectionProtocol = class(TObject)
  Server-side implementation of a proxy connection to any SynDB engine
- this default implementation will send the data without compression, digital signature, nor
  encryption
- inherit from this class to customize the transmission layer content

constructor Create(aAuthenticate: TSynAuthenticationAbstract); reintroduce;
  Initialize a protocol, with a given authentication scheme
- if no authentication is given, none will be processed

destructor Destroy; override;
  Release associated authentication class

property Authenticate: TSynAuthenticationAbstract read GetAuthenticate write fAuthenticate;
  The associated authentication information
- you can manage users via AuthenticateUser/DisauthenticateUser methods

TSQLDBRemoteConnectionProtocol = class(TSQLDBProxyConnectionProtocol)
  Server-side implementation of a remote connection to any SynDB engine
- implements digitally signed SynLZ-compressed binary message format, with simple symmetric
  encryption, as expected by SynDBRemote.pas
TSQLDBConnection = class(TObject)

Abstract connection created from TSQLDBConnectionProperties
- more than one TSQLDBConnection instance can be run for the same TSQLDBConnectionProperties

constructor Create(aProperties: TSQLDBConnectionProperties); virtual;
  Connect to a specified database engine

destructor Destroy; override;
  Release memory and connection

function IsConnected: boolean; virtual; abstract;
  Return TRUE if Connect has been already successfully called

function NewStatement: TSQLDBStatement; virtual; abstract;
  Initialize a new SQL query statement for the given connection
  - the caller should free the instance after use

function NewStatementPrepared(const aSQL: RawUTF8; ExpectResults: Boolean; RaiseExceptionOnError: Boolean=false; AllowReconnect: Boolean=true): ISQLDBStatement; virtual;
  Initialize a new SQL query statement for the given connection
  - this default implementation will call the NewStatement method, and implement handle statement caching is UseCache=true - in this case, the TSQLDBStatement.Reset method shall have been overridden to allow binding and execution of the very same prepared statement
  - the same aSQL can cache up to 9 statements in this TSQLDBConnection
  - this method should return a prepared statement instance on success
  - on error, if RaiseExceptionOnError=false (by default), it returns nil and you can check LastErrorMessage and LastErrorException properties to retrieve corresponding error information
  - if TSQLDBConnectionProperties.ReconnectAfterConnectionError is set, any connection error will be trapped, unless AllowReconnect is false
  - on error, if RaiseExceptionOnError=true, an exception is raised

function NewTableFromRows(const TableName: RawUTF8; Rows: TSQLDBStatement; WithinTransaction: boolean; ColumnForcedTypes: TSQLDBFieldTypeDynArray=nil): integer;
  Direct export of a DB statement rows into a new table of this database
  - the corresponding table will be created within the current connection, if it does not exist
  - if the column types are not set, they will be identified from the first row of data
  - INSERTs will be nested within a transaction if WithinTransaction is TRUE
  - will raise an Exception in case of error

procedure Commit; virtual;
  Commit changes of a Transaction for this connection
  - StartTransaction method must have been called before
  - this default implementation will check and set TransactionCount
procedure Connect; virtual;
    
    Connect to the specified database
    - should raise an Exception on error
    - this default implementation will notify OnProgress callback for successful re-connection: it should be called in overridden methods AFTER actual connection process

procedure Disconnect; virtual;
    
    Stop connection to the specified database
    - should raise an Exception on error
    - this default implementation will release all cached statements: so it should be called in overridden methods BEFORE actual disconnection

procedure RemoteProcessMessage(const Input: RawByteString; out Output: RawByteString; Protocol: TSQLDBProxyConnectionProtocol); virtual;
    
    Server-side implementation of a remote connection to any SynDB engine
    - follow the compressed binary message format expected by the TSQLDBRemoteConnectionPropertiesAbstract.ProcessMessage method
    - any transmission protocol could call this method to execute the corresponding TSQLDBProxyConnectionCommand on the current connection

procedure Rollback; virtual;
    
    Discard changes of a Transaction for this connection
    - StartTransaction method must have been called before
    - this default implementation will check and set TransactionCount

procedure StartTransaction; virtual;
    
    Begin a Transaction for this connection
    - this default implementation will check and set TransactionCount

property Connected: boolean read IsConnected;
    
    Returns TRUE if the connection was set

property InTransaction: boolean read GetInTransaction;
    
    TRUE if StartTransaction has been called
    - check if TransactionCount>0

property LastErrorException: ExceptClass read fErrorException;
    
    Some error exception, e.g. during execution of NewStatementPrepared

property LastErrorMessage: RawUTF8 read fErrorMessage write fErrorMessage;
    
    Some error message, e.g. during execution of NewStatementPrepared

property LastErrorWasAboutConnection: boolean read GetLastErrorWasAboutConnection;
    
    TRUE if last error is a broken connection, e.g. during execution of NewStatementPrepared
    - i.e. LastErrorException/LastErrorMessage concerns the database connection
    - will use TSQLDBConnectionProperties.ExceptionIsAboutConnection virtual method

    
    This event handler will be called during all process
    - can be used e.g. to change the desktop cursor
    - by default, will follow TSQLDBConnectionProperties.OnProcess property

property Properties: TSQLDBConnectionProperties read fProperties;
    
    The associated database properties
property RollbackOnDisconnect: Boolean read fRollbackOnDisconnect write fRollbackOnDisconnect;

Defines if Disconnect shall Rollback any pending transaction
- some engines executes a COMMIT when the client is disconnected, others do raise an exception: this parameter ensures that any pending transaction is roll-backed before disconnection
- is set to TRUE by default

property ServerDateTime: TDateTime read GetServerDateTime;

The current Date and Time, as retrieved from the server
- note that this value is the DB_SERVERTIME[] constant SQL value, so will most likely return a local time, not an UTC time
- this property will return the value as regular TDateTime

property ServerTimestamp: TTimeLog read GetServerTimestamp;

The current Date and Time, as retrieved from the server
- note that this value is the DB_SERVERTIME[] constant SQL value, so will most likely return a local time, not an UTC time
- this property will return the timestamp in TTimeLog / TTimeLogBits / Int64 value

property ServerTimestampAtConnection: TDateTime read fServerTimestampAtConnection;

The time returned by the server when the connection occurred

property TotalConnectionCount: integer read fTotalConnectionCount;

Number of successful connections for this instance
- can be greater than 1 in case of re-connection via Disconnect/Connect

property TransactionCount: integer read fTransactionCount;

Number of nested StartTransaction calls
- equals 0 if no transaction is active

TSQLDBStatement = class(TInterfacedObject)

Generic abstract class to implement a prepared SQL query
- inherited classes should implement the DB-specific connection in its overridden methods, especially Bind*(), Prepare(), ExecutePrepared, Step() and Column*() methods

constructor Create(aConnection: TSQLDBConnection); virtual;

Create a statement instance

function BoundCursor(Param: Integer): ISQLDBRows; virtual;

Return a special CURSOR parameter content as a SynDB result set
- this method is not about a column, but a parameter defined with BindCursor() before method execution
- Cursors are not handled internally by mORMot, but some databases (e.g. Oracle) usually use such structures to get data from stored procedures
- this method allow direct access to the data rows after execution
- this default method will raise an exception about unexpected behavior

function ColumnBlob(Col: integer): RawByteString; overload; virtual; abstract;

Return a Column as a blob value of the current Row, first Col is 0
function ColumnBlob(const ColName: RawUTF8): RawByteString; overload;
Return a Column as a blob value of the current Row, from a supplied column name

function ColumnBlobBytes(const ColName: RawUTF8): TBytes; overload;
Return a Column as a blob value of the current Row, from a supplied column name

function ColumnBlobBytes(Col: integer): TBytes; overload; virtual;
Return a Column as a blob value of the current Row, first Col is 0
- this function will return the BLOB content as a TBytes
- this default virtual method will call ColumnBlob()

function ColumnCount: integer;
The column/field count of the current Row

function ColumnCurrency(const ColName: RawUTF8): currency; overload;
Return a Column currency value of the current Row, from a supplied column name

function ColumnCurrency(Col: integer): currency; overload; virtual; abstract;
Return a Column currency value of the current Row, first Col is 0

function ColumnCursor(const ColName: RawUTF8): ISQLDBRows; overload;
Return a special CURSOR Column content as a SynDB result set
- Cursors are not handled internally by mORMot, but some databases (e.g. Oracle) usually use such structures to get data from stored procedures
- such columns are mapped as ftNull internally - so this method is the only one giving access to the data rows
- this default method will raise an exception about unexpected behavior

function ColumnCursor(Col: integer): ISQLDBRows; overload; virtual;
Return a special CURSOR Column content as a SynDB result set
- Cursors are not handled internally by mORMot, but some databases (e.g. Oracle) usually use such structures to get data from stored procedures
- such columns are mapped as ftNull internally - so this method is the only one giving access to the data rows
- this default method will raise an exception about unexpected behavior

function ColumnDateTime(const ColName: RawUTF8): TDateTime; overload;
Return a Column date and time value of the current Row, from a supplied column name

function ColumnDateTime(Col: integer): TDateTime; overload; virtual; abstract;
Return a Column date and time value of the current Row, first Col is 0

function ColumnDouble(const ColName: RawUTF8): double; overload;
Return a Column floating point value of the current Row, from a supplied column name

function ColumnDouble(Col: integer): double; overload; virtual; abstract;
Return a Column floating point value of the current Row, first Col is 0

function ColumnIndex(const aColumnName: RawUTF8): integer; virtual; abstract;
Returns the Column index of a given Column name
- Columns numeration (i.e. Col value) starts with 0
- returns -1 if the Column name is not found (via case insensitive search)
### Synopsis mORMot Framework

**Software Architecture Design 1.18**  
**Date:** September 16, 2020

---

**function** `ColumnInt(Col: integer): Int64; overload; virtual; abstract;`  
*Return a Column integer value of the current Row, first Col is 0*

**function** `ColumnInt(const ColName: RawUTF8): Int64; overload;`  
*Return a Column integer value of the current Row, from a supplied column name*

**function** `ColumnName(Col: integer): RawUTF8; virtual; abstract;`  
*The Column name of the current Row*
- Columns numeration (i.e. Col value) starts with 0
- it's up to the implementation to ensure than all column names are unique

**function** `ColumnNull(Col: integer): boolean; virtual; abstract;`  
*Returns TRUE if the column contains NULL*

**function** `ColumnsToSQLInsert(const TableName: RawUTF8; var Fields: TSQLDBColumnCreateDynArray): RawUTF8; virtual;`  
*Compute the SQL INSERT statement corresponding to this columns row*
- and populate the Fields[] array with columns information (type and name)
- if the current column value is NULL, will return ftNull: it is up to the caller to set the proper field type
- the SQL statement is prepared with bound parameters, e.g.
  ```sql```
  insert into TableName (Col1,Col2) values (?,N)
  ```
  - used e.g. to convert some data on the fly from one database to another, via the TSQLDBConnection.NewTableFromRows method

**function** `ColumnString(Col: integer): string; overload; virtual;`  
*Return a Column text value as generic VCL string of the current Row, first Col is 0*
- this default implementation will call ColumnUTF8

**function** `ColumnString(const ColName: RawUTF8): string; overload;`  
*Return a Column text value as generic VCL string of the current Row, from a supplied column name*

**function** `ColumnTimestamp(const ColName: RawUTF8): TTimeLog; overload;`  
*Return a column date and time value of the current Row, from a supplied column name*
- call ColumnDateTime or ColumnUTF8 to convert into TTimeLogBits/Int64 time stamp from a TDateTime or text

**function** `ColumnTimestamp(Col: integer): TTimeLog; overload;`  
*Return a column date and time value of the current Row, first Col is 0*
- call ColumnDateTime or ColumnUTF8 to convert into TTimeLogBits/Int64 time stamp from a TDateTime or text

**function** `ColumnToVariant(Col: integer; var Value: Variant): TSQLDBFieldType; virtual;`  
*Return a Column as a variant, first Col is 0*
- this default implementation will call Column*() method above
- a ftUTF8 TEXT content will be mapped into a generic WideString variant for pre-Unicode version of Delphi, and a generic UnicodeString (=string) since Delphi 2009: you may not loose any data during charset conversion
- a ftBlob BLOB content will be mapped into a TBlobData AnsiString variant
function ColumnType(Col: integer; FieldSize: PInteger=nil): TSQLDBFieldType;
virtual; abstract;

The Column type of the current Row
- FieldSize can be set to store the size in chars of a ftUTF8 column (0 means BLOB kind of TEXT column)

function ColumnUTF8(Col: integer): RawUTF8; overload; virtual; abstract;

Return a Column UTF-8 encoded text value of the current Row, first Col is 0

function ColumnUTF8(const ColName: RawUTF8): RawUTF8; overload;

Return a Column UTF-8 encoded text value of the current Row, from a supplied column name

function ColumnVariant(Col: integer): Variant; overload;

Return a Column as a variant, first Col is 0
- this default implementation will call ColumnToVariant() method
- a ftUTF8 TEXT content will be mapped into a generic WideString variant for pre-Unicode version of Delphi, and a generic UnicodeString (=string) since Delphi 2009: you may not loose any data during charset conversion
- a ftBlob BLOB content will be mapped into a TBlobData AnsiString variant

function ColumnVariant(const ColName: RawUTF8): Variant; overload;

Return a Column as a variant, from a supplied column name

function FetchAllAsJSON(Expanded: boolean; ReturnedRowCount: PPtrInt=nil): RawUTF8;

Return all rows content as a JSON string
- JSON data is retrieved with UTF-8 encoding
- if Expanded is true, JSON data is an array of objects, for direct use with any Ajax or .NET client:
  [ {"col1":val11, "col2": "val12"},{"col1":val21,...] 
- if Expanded is false, JSON data is serialized (used in TSQLTableJSON)
  { "FieldCount":1, "Values":["col1","col2",val11,"val12",val21,...] } 
- BLOB field value is saved as Base64, in the "\uFFF0base64encodedbinary" format and contains true BLOB data
- if ReturnedRowCount points to an integer variable, it will be filled with the number of row data returned (excluding field names)
- similar to corresponding TSQLRequest.Execute method in SynSQLite3 unit

function FetchAllToBinary(Dest: TStream; MaxRowCount: cardinal=0; DataRowPosition: PCardinalDynArray=nil): cardinal;
virtual;

Append all rows content as binary stream
- will save the column types and name, then every data row in optimized binary format (faster and smaller than JSON)
- you can specify a LIMIT for the data extent (default 0 meaning all data)
- generates the format expected by TSQldbProxyStatement
function FetchAllToCSVValues(Dest: TStream; Tab: boolean; CommaSep: AnsiChar = ','; AddBOM: boolean = true): PtrInt;

Append all rows content as a CSV stream
- CSV data is added to the supplied TStream, with UTF-8 encoding
- if Tab=TRUE, will use TAB instead of ',' between columns
- you can customize the ',' separator - use e.g. the global ListSeparator variable (from SysUtils) to reflect the current system definition (some country use ',' as decimal separator, for instance our "douce France")
- AddBOM will add a UTF-8 Byte Order Mark at the beginning of the content
- BLOB fields will be appended as "blob" with no data
- returns the number of row data returned

function FetchAllToJSON(JSON: TStream; Expanded: boolean): PtrInt;

Append all rows content as a JSON stream
- JSON data is added to the supplied TStream, with UTF-8 encoding
- if Expanded is true, JSON data is an array of objects, for direct use with any Ajax or .NET client:
  
  ```
  [{ "col1":val11,"col2": "val12"},{"col1":val21,... ]
  ```
- if Expanded is false, JSON data is serialized (used in TSQLTableJSON)
  ```
  { "FieldCount":1,"Values":["col1","col2",val11,"val12",val21,...] }
  ```
- BLOB field value is saved as Base64, in the "\uFFF0base64encodedbinary" format and contains true BLOB data
- similar to corresponding TSQLRequest.Execute method in SynSQLite3 unit
- returns the number of row data returned (excluding field names)
- warning: TSQLRestStorageExternal.EngineRetrieve in mORMotDB unit expects the Expanded=true format to return '{[...]}'

function ParamToVariant(Param: Integer; var Value: Variant; CheckIsOutParameter: boolean = true): TSQLDBFieldType;

Retrieve the parameter content, after SQL execution
- the leftmost SQL parameter has an index of 1
- to be used e.g. with stored procedures:
  ```
  query := 'BEGIN TEST_PKG.DUMMY(?, ?, ?, ?, ?); END;';
  stmt := Props.NewThreadSafeStatementPrepared(query, false);
  stmt.Bind(1, in1, paramIn);
  stmt.BindTextU(2, in2, paramIn);
  stmt.BindTextU(3, in3, paramIn);
  stmt.BindTextS(4, '', paramOut); // to be retrieved with out1: string
  stmt.Bind(5, 0, paramOut); // to be retrieved with out2: integer
  stmt.ExecutePrepared;
  stmt.ParamToVariant(4, out1, true);
  stmt.ParamToVariant(5, out2, true);
  ```
- the parameter should have been bound with IO=paramOut or IO=paramInOut if CheckIsOutParameter is TRUE
- this implementation just check that Param is correct: overridden method should fill Value content
function RowData: Variant; virtual;

Create a TSQLDBRowVariantType able to access any field content via late binding
- i.e. you can use Data.Name to access the 'Name' column of the current row
- this Variant will point to the corresponding TSQLDBStatement instance, so it's not necessary to retrieve its value for each row
- typical use is:
  var Row: Variant;
  (...)  
  with MyConnProps.Execute('select * from table where name=?', [aName]) do begin
    Row := RowData;
    while Step do
      writeln(Row.FirstName, Row.BirthDate);
  end;

function Step(SeekFirst: boolean=false): boolean; virtual; abstract;

After a statement has been prepared via Prepare() + ExecutePrepared() or Execute(), this method must be called one or more times to evaluate it
- you shall call this method before calling any Column*() methods
- return TRUE on success, with data ready to be retrieved by Column*()
- return FALSE if no more row is available (e.g. if the SQL statement is not a SELECT but an UPDATE or INSERT command)
- access the first or next row of data from the SQL Statement result: if SeekFirst is TRUE, will put the cursor on the first row of results, otherwise, it will fetch one row of data, to be called within a loop
- should raise an Exception on any error
- typical use may be (see also e.g. the mORMotDB unit):
  var Query: ISQLDBStatement;
  begin
where AccountNumber like ?', ['AW000001%'], true);
    if Query<>nil then begin
      assert(SameTextU(Query.ColumnName(0), 'AccountNumber'));
      while Query.Step do // loop through all matching data rows
        assert(Copy(Query.ColumnUTF8(0), 1, 8)='AW000001');
      Query.ReleaseRows;
    end;
  end;

function UpdateCount: integer; virtual;

Gets a number of updates made by latest executed statement
- default implementation returns 0

procedure Bind(Param: Integer; Value: double; IO: TSQLDBParamInOutType=paramIn); overload; virtual; abstract;

Bind a double value to a parameter
- the leftmost SQL parameter has an index of 1
procedure Bind(const Params: array of const; IO: TSQLDBParamInOutType=paramIn); overload; virtual;

Bind an array of const values
- parameters marked as ? should be specified as method parameter in Params[]
- BLOB parameters can be bound with this method, when set after encoding via
  BinToBase64WithMagic() call
- TDateTime parameters can be bound with this method, when encoded via a DateToSQL() or
  DateTimeToSQL() call
- any variant parameter will be bound with BindVariant(i,Variant^,true,IO) i.e. with
  DataIsBlob=true
- this default implementation will call corresponding Bind*() method

procedure Bind(Param: Integer; ParamType: TSQLDBFieldType; const Value: RawUTF8; ValueAlreadyUnquoted: boolean; IO: TSQLDBParamInOutType=paramIn); overload; virtual;

Bind one RawUTF8 encoded value
- the leftmost SQL parameter has an index of 1
- the value should match the BindArray() format, i.e. be stored as in SQL (i.e. number, 'quoted
  string', 'YYYY-MM-DD hh:mm:ss', null) - e.g. as computed by TJSONObjectDecoder.Decode()

procedure Bind(Param: Integer; const Data: TSQLVar; IO: TSQLDBParamInOutType=paramIn); overload; virtual;

Bind one TSQLVar value
- the leftmost SQL parameter has an index of 1
- this default implementation will call corresponding Bind*() method

procedure Bind(Param: Integer; Value: Int64; IO: TSQLDBParamInOutType=paramIn); overload; abstract;

Bind an integer value to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindArray(Param: Integer; const Values: array of double); overload; virtual;

Bind an array of double values to a parameter
- the leftmost SQL parameter has an index of 1
- this default implementation will raise an exception if the engine does not support array binding

procedure BindArray(Param: Integer; ParamType: TSQLDBFieldType; const Values: TRawUTF8DynArray; ValuesCount: integer); overload; virtual;

Bind an array of values to a parameter
- the leftmost SQL parameter has an index of 1
- values are stored as in SQL (i.e. number, 'quoted string', 'YYYY-MM-DD hh:mm:ss', null)
- this default implementation will raise an exception if the engine does not support array binding

procedure BindArray(Param: Integer; const Values: array of Int64); overload; virtual;

Bind an array of integer values to a parameter
- the leftmost SQL parameter has an index of 1
- this default implementation will raise an exception if the engine does not support array binding
**procedure BindArray(Param: Integer; const Values: array of RawUTF8); overload; virtual;**

*Bind an array of RawUTF8 values to a parameter*
- the leftmost SQL parameter has an index of 1
- values are stored as in SQL (i.e. ‘quoted string’)
- this default implementation will raise an exception if the engine does not support array binding

**procedure BindArrayCurrency(Param: Integer; const Values: array of currency); virtual;**

*Bind an array of currency values to a parameter*
- the leftmost SQL parameter has an index of 1
- this default implementation will raise an exception if the engine does not support array binding

**procedure BindArrayDateTime(Param: Integer; const Values: array of TDateTime); virtual;**

*Bind an array of TDateTime values to a parameter*
- the leftmost SQL parameter has an index of 1
- values are stored as in SQL (i.e. 'YYYY-MM-DD hh:mm:ss')
- this default implementation will raise an exception if the engine does not support array binding

**procedure BindBlob(Param: Integer; Data: pointer; Size: integer; IO: TSQLDBParamInOutType=paramIn); overload; virtual; abstract;**

*Bind a Blob buffer to a parameter*
- the leftmost SQL parameter has an index of 1

**procedure BindBlob(Param: Integer; const Data: RawByteString; IO: TSQLDBParamInOutType=paramIn); overload; virtual; abstract;**

*Bind a Blob buffer to a parameter*
- the leftmost SQL parameter has an index of 1

**procedure BindCurrency(Param: Integer; Value: currency; IO: TSQLDBParamInOutType=paramIn); overload; virtual; abstract;**

*Bind a currency value to a parameter*
- the leftmost SQL parameter has an index of 1

**procedure BindCursor(Param: integer); virtual;**

*Bind a special CURSOR parameter to be returned as a SynDB result set*
- Cursors are not handled internally by mORMot, but some databases (e.g. Oracle) usually use such structures to get data from stored procedures
- such parameters are mapped as ftUnknown
- use BoundCursor() method to retrieve the corresponding ISQLDBRows after execution of the statement
- this default method will raise an exception about unexpected behavior

**procedure BindDateTime(Param: Integer; Value: TDateTime; IO: TSQLDBParamInOutType=paramIn); overload; virtual; abstract;**

*Bind a TDateTime value to a parameter*
- the leftmost SQL parameter has an index of 1
procedure BindFromRows(const Fields: TSQLDBFieldTypeDynArray; Rows: TSQLDBStatement);

Bind an array of fields from an existing SQL statement
- can be used e.g. after ColumnsToSQLInsert() method call for fast data conversion between tables

procedure BindNull(Param: Integer; IO: TSQLDBParamInOutType=paramIn; BoundType: TSQLDBFieldType=ftNull); virtual; abstract;

Bind a NULL value to a parameter
- the leftmost SQL parameter has an index of 1
- some providers (e.g. OleDB during MULTI INSERT statements) expect the proper column type to be set in BoundType, even for NULL values

procedure BindTextP(Param: Integer; Value: PUTF8Char; IO: TSQLDBParamInOutType=paramIn); overload; virtual; abstract;

Bind a UTF-8 encoded buffer text (#0 ended) to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindTextS(Param: Integer; const Value: string; IO: TSQLDBParamInOutType=paramIn); overload; virtual; abstract;

Bind a UTF-8 encoded string to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindTextU(Param: Integer; const Value: RawUTF8; IO: TSQLDBParamInOutType=paramIn); overload; virtual; abstract;

Bind a UTF-8 encoded string to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindTextW(Param: Integer; const Value: WideString; IO: TSQLDBParamInOutType=paramIn); overload; virtual; abstract;

Bind a UTF-8 encoded string to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindVariant(Param: Integer; const Data: Variant; DataIsBlob: boolean; IO: TSQLDBParamInOutType=paramIn); virtual;

Bind a Variant value to a parameter
- the leftmost SQL parameter has an index of 1
- will call all virtual Bind*() methods from the Data type
- if DataIsBlob is TRUE, will call BindBlob(RawByteString(Data)) instead of BindTextW(WideString(Variant)) - used e.g. by TQuery.AsBlob/AsBytes

procedure ColumnBlobFromStream(const ColName: RawUTF8; Stream: TStream); overload;

Write a blob Column into the Stream parameter
- expected to be used with 'SELECT .. FOR UPDATE' locking statements

procedure ColumnBlobFromStream(Col: integer; Stream: TStream); overload; virtual;

Write a blob Column into the Stream parameter
- expected to be used with 'SELECT .. FOR UPDATE' locking statements
- default implementation will through an exception, since it is highly provider-specific; SynDBOracle e.g. implements it properly

procedure ColumnBlobToStream(const ColName: RawUTF8; Stream: TStream); overload;

Read a blob Column into the Stream parameter
procedure ColumnBlobToStream(Col: integer; Stream: TStream); overload; virtual;

Read a blob Column into the Stream parameter
- default implementation will just call ColumnBlob(), whereas some providers (like SynDBOracle)
  may implement direct support

procedure ColumnsToBinary(W: TFileBufferWriter; Null: pointer; const ColTypes: TSQLDBFieldTypeDynArray); virtual;

Append current row content as binary stream
- will save one data row in optimized binary format (if not in Null)
- virtual method called by FetchAllToBinary()
- follows the format expected by TSQLDBProxyStatement

procedure ColumnsToJSON(WR: TJSONWriter); virtual;

Append all columns values of the current Row to a JSON stream
- will use WR.Expand to guess the expected output format
- this default implementation will call Column*() methods above, but you should also implement
  a custom version with no temporary variable
- BLOB field value is saved as Base64, in the "\uFFF0base64encodedbinary" format and contains
  true BLOB data (unless ForceBlobAsNull property was set)

procedure ColumnToSQLVar(Col: Integer; var Value: TSQLVar; var Temp: RawByteString); virtual;

Return a Column as a TSQLVar value, first Col is 0
- the specified Temp variable will be used for temporary storage of svtUTF8/svtBlob values

procedure Execute(const aSQL: RawUTF8; ExpectResults: Boolean; const Params: array of const); overload;

Prepare and Execute an UTF-8 encoded SQL statement
- parameters marked as ? should be specified as method parameter in Params[]
- BLOB parameters could not be bound with this method, but need an explicit call to BindBlob() method
- if ExpectResults is TRUE, then Step() and Column*() methods are available to retrieve the data rows
- should raise an Exception on any error
- this method will bind parameters, then call Execute() virtual method

procedure Execute(const SQLFormat: RawUTF8; ExpectResults: Boolean; const Args, Params: array of const); overload;

Prepare and Execute an UTF-8 encoded SQL statement
- parameters marked as % will be replaced by Args[] value in the SQL text
- parameters marked as ? should be specified as method parameter in Params[]
- so could be used as such, mixing both % and ? parameters:
  Statement.Execute('SELECT % FROM % WHERE RowID=?',true,[FieldName,TableName],[ID])
- BLOB parameters could not be bound with this method, but need an explicit call to BindBlob() method
- if ExpectResults is TRUE, then Step() and Column*() methods are available to retrieve the data rows
- should raise an Exception on any error
- this method will bind parameters, then call Execute() virtual method
procedure Execute(const aSQL: RawUTF8; ExpectResults: Boolean); overload;

Prepare and Execute an UTF-8 encoded SQL statement
- parameters marked as ? should have been already bound with Bind*() functions above
- if ExpectResults is TRUE, then Step() and Column*() methods are available to retrieve the data rows
- should raise an Exception on any error
- this method will call Prepare then ExecutePrepared methods

procedure ExecutePrepared; virtual;

Execute a prepared SQL statement
- parameters marked as ? should have been already bound with Bind*() functions
- should raise an Exception on any error
- this void default implementation will call set fConnection.fLastAccess

procedure ExecutePreparedAndFetchAllAsJSON(Expanded: boolean; out JSON: RawUTF8); virtual;

Execute a prepared SQL statement and return all rows content as a JSON string
- JSON data is retrieved with UTF-8 encoding
- if Expanded is true, JSON data is an array of objects, for direct use with any Ajax or .NET client:

```plaintext
[ { "col1":val11, "col2": "val12" }, { "col1":val21, ... }
```
- if Expanded is false, JSON data is serialized (used in TSQLTableJSON)

```plaintext
{ "FieldCount":1, "Values": [ "col1", "col2", val11, "val12", val21, ... ] }
```
- BLOB field value is saved as Base64, in the "\uFFF0base64encodedbinary" format and contains true BLOB data
- this virtual implementation calls ExecutePrepared then FetchAllAsJSON()

procedure Prepare(const aSQL: RawUTF8; ExpectResults: Boolean); overload; virtual;

Prepare an UTF-8 encoded SQL statement
- parameters marked as ? will be bound later, before ExecutePrepared call
- if ExpectResults is TRUE, then Step() and Column*() methods are available to retrieve the data rows
- should raise an Exception on any error
- this default implementation will just store aSQL content and the ExpectResults parameter, and connect to the remote server is was not already connected

procedure ReleaseRows; virtual;

Release cursor memory and resources once Step loop is finished
- this method call is optional, but is better be used if the ISQLDBRows statement from taken from cache, and returned a lot of content which may still be in client (and server) memory
- override to free cursor memory when ISQLDBStatement is back in cache

procedure Reset; virtual;

Reset the previous prepared statement
- some drivers expect an explicit reset before binding parameters and executing the statement another time
- this default implementation will just do nothing

procedure RowDocVariant(out aDocument: variant; aOptions: TDocVariantOptions=JSON_OPTIONS_FAST); virtual;

Create a TDocVariant custom variant containing all columns values
- will create a "fast" TDocVariant object instance with all fields
property CacheIndex: integer read fCacheIndex;
   Low-level access to the statement cache index, after a call to Prepare()
   - contains >= 0 if the database supports prepared statement cache (Oracle, Postgres) and query
   plan is cached; contains -1 in other cases

property Connection: TSQLDBConnection read fConnection;
   The associated database connection

property CurrentRow: Integer read fCurrentRow;
   The current row after Execute/Step call, corresponding to Column*() methods
   - contains 0 before initial Step call, or a number >=1 during data retrieval

property SQL: RawUTF8 read fSQL;
   The prepared SQL statement, as supplied to Prepare() method

property SQLCurrent: RawUTF8 read GetSQLCurrent;
   The prepared SQL statement, in its current state
   - if statement is prepared, then equals SQLPrepared, otherwise, contains the raw SQL property
   content
   - used internally by the implementation units, e.g. for errors logging

property SQLLogTimer: TPrecisionTimer read fSQLLogTimer;
   Low-level access to the Timer used for last DB operation

property SQLPrepared: RawUTF8 read fSQLPrepared;
   After a call to Prepare(), contains the query text to be passed to the DB
   - depending on the DB, parameters placeholders are replaced by ?, :1, $1 etc
   - this SQL is ready to be used in any DB tool, e.g. to check the real execution plan/timing

property SQLWithInlinedParams: RawUTF8 read GetSQLWithInlinedParams;
   The prepared SQL statement, with all '?' changed into the supplied parameter values
   - such statement query plan usually differ from a real execution plan for prepared statements
   with parameters - see SQLPrepared property instead

property StripSemicolon: boolean read fStripSemicolon write fStripSemicolon;
   Strip last semicolon in query
   - expectation may vary, depending on the SQL statement and the engine
   - default is true

property TotalRowsRetrieved: Integer read fTotalRowsRetrieved;
   The total number of data rows retrieved by this instance
   - is not reset when there is no more row of available data (Step returns false), or when Step() is
   called with SeekFirst=true

TSQLDBConnectionThreadSafe = class(TSQLDBConnection)
   Abstract connection created from TSQLDBConnectionProperties
   - this overridden class will defined an hidden thread ID, to ensure that one connection will be
   create per thread
   - e.g. OleDb, ODBC and Oracle connections will inherit from this class

TSQLDBConnectionPropertiesThreadSafe = class(TSQLDBConnectionProperties)
   Connection properties which will implement an internal Thread-Safe connection pool
**constructor** Create(const aServerName, aDatabaseName, aUserID, aPassword: RawUTF8); **override**

*Initialize the properties*
- this overridden method will initialize the internal per-thread connection pool

**destructor** Destroy; **override**

*Release related memory, and all per-thread connections*

**function** ThreadSafeConnection: TSQLDBConnection; **override**

*Get a thread-safe connection*
- this overridden implementation will define a per-thread TSQLDBConnection connection pool, via an internal pool

**procedure** ClearConnectionPool; **override**

*Release all existing connections*
- this overridden implementation will release all per-thread TSQLDBConnection internal connection pool
- warning: no connection shall still be used on the background (e.g. in multi-threaded applications), or some unexpected border effects may occur

**procedure** EndCurrentThread; **virtual**

*You can call this method just before a thread is finished to ensure that the associated Connection will be released*
- could be used e.g. in a try...finally block inside a TThread.Execute overridden method
- could be used e.g. to call CoUninitialize from thread in which CoInitialize was made, for instance via a method defined as such:

```pascal
procedure TMyServer.OnHttpThreadTerminate(Sender: TObject);
begin
  fMyConnectionProps.EndCurrentThread;
end;
```
- this method shall be called from the thread about to be terminated: e.g. if you call it from the main thread, it may fail to release resources
- within the mORMot server, mORMotDB unit will call this method for every terminating thread created for TSQLRestServerNamedPipeResponse or TSQLHttpServer multi-thread process

**property** ThreadingMode: TSQLDBConnectionPropertiesThreadSafeThreadingMode **read** fThreadingMode **write** fThreadingMode;

*Set this property if you want to disable the per-thread connection pool*
- to be used e.g. in database embedded mode (SQLite3/FireBird), when multiple connections may break stability and decrease performance
- see TSQLDBConnectionPropertiesThreadSafeThreadingMode for the possible values

**TSQDBParam** = **packed** record

*A structure used to store a standard binding parameter*
- you can use your own internal representation of parameters (TODBStatement use its own TODBStatementParam type), but this type can be used to implement a generic parameter
- used e.g. by TSQLDBStatementWithParams as a dynamic array (and its inherited TSQLDBOracleStatement)
- don't change this structure, since it will be serialized as binary for TSQLDBProxyConnectionCommandExecute
VArray: TRawUTF8DynArray;

Storage used for bound array values
- number of items in array is stored in VInt64
- values are stored as in SQL (i.e. number, 'quoted string', 'YYYY-MM-DD hh:mm:ss', null)

VData: RawByteString;

Storage used for TEXT (ftUTF8) and BLOB (ftBlob) values
- ftBlob are stored as RawByteString
- ftUTF8 are stored as RawUTF8
- sometimes, may be ftInt64 or ftCurrency provided as SQLT_AVC text, or ftDate value converted to SQLT_TIMESTAMP

VDBType: word;

Used e.g. by TSQLDBOracleStatement

VINOut: TSQLDBParamInOutType;

Define if parameter can be retrieved after a stored procedure execution

VInt64: Int64;

Storage used for ftInt64, ftDouble, ftDate and ftCurrency value

VType: TSQLDBFieldType;

The column/parameter Value type

TSQLDBStatementWithParams = class(TSQLDBStatement)

Generic abstract class handling prepared statements with binding
- will provide protected fields and methods for handling standard TSQLDBParam parameters

constructor Create(aConnection: TSQLDBConnection); override;

Create a statement instance
- this overridden version will initialize the internal fParam* fields

function ParamToVariant(Param: Integer; var Value: Variant; CheckIsOutParameter: boolean=true): TSQLDBFieldType; override;

Retrieve the parameter content, after SQL execution
- the leftmost SQL parameter has an index of 1
- to be used e.g. with stored procedures
- this overridden function will retrieve the value stored in the protected fParams[] array: the ExecutePrepared method should have updated its content as expected

procedure Bind(Param: Integer; Value: double; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a double value to a parameter
- the leftmost SQL parameter has an index of 1
- raise an Exception on any error

procedure Bind(Param: Integer; Value: Int64; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind an integer value to a parameter
- the leftmost SQL parameter has an index of 1
- raise an Exception on any error
procedure BindArray(Param: Integer; const Values: array of double); overload; override;

*Bind an array of double values to a parameter*
- the leftmost SQL parameter has an index of 1
- this default implementation will raise an exception if the engine does not support array binding
- this default implementation will call BindArray() after conversion into RawUTF8 items, stored in TSQLDBParam.VArray

procedure BindArray(Param: Integer; const Values: array of Int64); overload; override;

*Bind an array of integer values to a parameter*
- the leftmost SQL parameter has an index of 1
- this default implementation will call BindArray() after conversion into RawUTF8 items, stored in TSQLDBParam.VArray

procedure BindArray(Param: Integer; const Values: array of RawUTF8); overload; override;

*Bind an array of RawUTF8 values to a parameter*
- the leftmost SQL parameter has an index of 1
- values are stored as 'quoted string'
- this default implementation will raise an exception if the engine does not support array binding

procedure BindArray(Param: Integer; ParamType: TSQLDBFieldType; const Values: TRawUTF8DynArray; ValuesCount: integer); overload; override;

*Bind an array of values to a parameter using OCI bind array feature*
- the leftmost SQL parameter has an index of 1
- values are stored as in SQL (i.e. number, 'quoted string', 'YYYY-MM-DD hh:mm:ss', null)
- values are stored as in SQL (i.e. 'YYYY-MM-DD hh:mm:ss')

procedure BindArrayCurrency(Param: Integer; const Values: array of currency); override;

*Bind an array of currency values to a parameter*
- the leftmost SQL parameter has an index of 1
- this default implementation will raise an exception if the engine does not support array binding
- this default implementation will call BindArray() after conversion into RawUTF8 items, stored in TSQLDBParam.VArray

procedure BindArrayDateTime(Param: Integer; const Values: array of TDateTime); override;

*Bind an array of TDateTime values to a parameter*
- the leftmost SQL parameter has an index of 1
- values are stored as in SQL (i.e. 'YYYY-MM-DD hh:mm:ss')
- this default implementation will raise an exception if the engine does not support array binding
- this default implementation will call BindArray() after conversion into RawUTF8 items, stored in TSQLDBParam.VArray

procedure BindArrayRow(const aValues: array of const);

*Bind a set of parameters for further array binding*
- supplied parameters shall follow the BindArrayRowPrepare() supplied types (i.e. RawUTF8, Integer/Int64, double); you can also bind directly a TDateTime value if the corresponding binding has been defined as ftDate by BindArrayRowPrepare()
procedure BindArrayRowPrepare(const aParamTypes: array of TSQLDBFieldType; aExpectedMinimalRowCount: integer=0);

Start parameter array binding per-row process
- BindArray*() methods expect the data to be supplied "vertically": this method allow-per row binding
- call this method, then BindArrayRow() with the corresponding values for one statement row, then Execute to send the query

procedure BindBlob(Param: Integer; Data: pointer; Size: integer; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a Blob buffer to a parameter
- the leftmost SQL parameter has an index of 1
- raise an Exception on any error

procedure BindBlob(Param: Integer; const Data: RawByteString; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a Blob buffer to a parameter
- the leftmost SQL parameter has an index of 1
- raise an Exception on any error

procedure BindCurrency(Param: Integer; Value: currency; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a currency value to a parameter
- the leftmost SQL parameter has an index of 1
- raise an Exception on any error

procedure BindDateTime(Param: Integer; Value: TDateTime; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a TDateTime value to a parameter
- the leftmost SQL parameter has an index of 1
- raise an Exception on any error

procedure BindFromRows(Rows: TSQLDBStatement); virtual;

Bind an array of fields from an existing SQL statement for array binding
- supplied Rows columns shall follow the BindArrayRowPrepare() supplied types (i.e. RawUTF8, Integer/Int64, double, date)
- can be used e.g. after ColumnsToSQLInsert() method call for fast data conversion between tables

procedure BindNull(Param: Integer; IO: TSQLDBParamInOutType=paramIn; BoundType: TSQLDBFieldType=ftNull); override;

Bind a NULL value to a parameter
- the leftmost SQL parameter has an index of 1
- raise an Exception on any error
- some providers (only OleDB during MULTI INSERT statements, so never used in this class)
  expect the proper column type to be set in BoundType

procedure BindTextP(Param: Integer; Value: PUTF8Char; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a UTF-8 encoded buffer text (#0 ended) to a parameter
- the leftmost SQL parameter has an index of 1
procedure BindTextS(Param: Integer; const Value: string; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a VCL string to a parameter
- the leftmost SQL parameter has an index of 1
- raise an Exception on any error

procedure BindTextU(Param: Integer; const Value: RawUTF8; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a UTF-8 encoded string to a parameter
- the leftmost SQL parameter has an index of 1
- raise an Exception on any error

procedure BindTextW(Param: Integer; const Value: WideString; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind an OLE WideString to a parameter
- the leftmost SQL parameter has an index of 1
- raise an Exception on any error

procedure ReleaseRows; override;

Release used memory
- this overridden implementation will free the fParams[] members (e.g. VData) but not the parameters themselves

procedure Reset; override;

Reset the previous prepared statement
- this overridden implementation will just do reset the internal fParams[]

TSQLDBStatementWithParamsAndColumns = class(TSQLDBStatementWithParams)

Generic abstract class handling prepared statements with binding and column description
- will provide protected fields and methods for handling both TSQLDBParam parameters and standard TSQLDBColumnProperty column description

constructor Create(aConnection: TSQLDBConnection); override;

Create a statement instance
- this overridden version will initialize the internal fColumn* fields

function ColumnIndex(const aColumnName: RawUTF8): integer; override;

Returns the Column index of a given Column name
- Columns numeration (i.e. Col value) starts with 0
- returns -1 if the Column name is not found (via case insensitive search)

function ColumnName(Col: integer): RawUTF8; override;

Retrieve a column name of the current Row
- Columns numeration (i.e. Col value) starts with 0
- it's up to the implementation to ensure than all column names are unique
**ColumnType**

*function* `ColumnType(Col: integer; FieldSize: PInteger=nil): TSQLDBFieldType; override;`

- The *Column type of the current Row* should handle specifically, for faster process and avoid any rounding issue, since currency is a standard OleDB type.
- `FieldSize` can be set to store the size in chars of a `ftUTF8` column (0 means BLOB kind of `TEXT` column - this implementation will store `fColumns[Col].ColumnValueDBSize` if `ColumnValueInlined=true`.

**Columns**

*property* `Columns: TSQLDBColumnPropertyDynArray read fColumns;`

*Direct access to the columns description*
- gives more details than the default `ColumnType()` function

**ESQLDBException** = *class*(ESynException)

*Generic Exception type, as used by the SynDB unit*

*constructor* `CreateUTF8(const Format: RawUTF8; const Args: array of const);`

*Constructor which will use FormatUTF8() instead of Format() - if the first Args[0] is a TSQLDBStatement class instance, the current SQL statement will be part of the exception message*

*property* `Statement: TSQLDBStatement read fStatement;`

*Associated TSQLDBStatement instance, if supplied as first parameter*

**ESQLDBRemote** = *class*(ESQLDBException)

*Exception raised during remote connection process*

**TSQldbProxyConnectionCommandExecute** = *packed record*

*Structure to embedd all needed parameters to execute a SQL statement*
- used for `cExecute`, `cExecuteToBinary`, `cExecuteToJSON` and `cExecuteToExpandedJSON` commands of `TSQLDBProxyConnectionProperties.Process()` - set by `TSQLDBProxyStatement.ParamsToCommand()` protected method

*ArrayCount: integer;*

*If input parameters expected BindArray() process*

*Force: set of (fBlobAsNull, fDateWithMS, fNoUpdateCount);*

*How server side would handle statement execution*
- `fBlobAsNull` and `fDateWithMS` do match `ForceBlobAsNull` and `ForceDateWithMS` `ISQLDBStatement` properties
- `fNoUpdateCount` avoids to call `ISQLDBStatement.UpdateCount` method, e.g. for performance reasons

*Params: TSQLDBParamDynArray;*

*Input parameters*
- trunked to the exact number of parameters

*SQL: RawUTF8;*

*The associated SQL statement*
**TSQLDBProxyConnectionPropertiesAbstract = class(TSQLDBConnectionProperties)**

*Implements a proxy-like virtual connection statement to a DB engine*
- will generate TSQLDBProxyConnection kind of connection

**destructor Destroy; override;**
- Will notify for proxy disconnection

**function IsCachable(P: PUTF8Char): boolean; override;**
- Determines if the SQL statement can be cached
  - always returns false, to force a new fake statement to be created

**function NewConnection: TSQLDBConnection; override;**
- Create a new TSQLDBProxyConnection instance
  - the caller is responsible of freeing this instance

**procedure GetFields(const aTableName: RawUTF8; out Fields: TSQLDBColumnDefineDynArray); override;**
- Retrieve the column/field layout of a specified table
  - calls Process(cGetFields,aTableName,Fields)

**procedure GetIndexes(const aTableName: RawUTF8; out Indexes: TSQLDBIndexDefineDynArray); override;**
- Retrieve the advanced indexed information of a specified Table
  - calls Process(cGetIndexes,aTableName,Indexes)

**procedure GetTableNames(out Tables: TRawUTF8DynArray); override;**
- Get all table names
  - this default implementation will use protected SQLGetTableNames virtual
  - calls Process(cGetTableNames,self,Tables)

**property HandleConnection: boolean read fHandleConnection write fHandleConnection;**
- Connect and Disconnect won’t really connect nor disconnect the remote connection
  - you can set this property to TRUE if you expect the remote connection by in synch with the remote proxy connection (should not be used in most cases, unless you are sure you have only one single client at a time)

**property StartTransactionTimeOut: Int64 read fStartTransactionTimeOut write fStartTransactionTimeOut;**
- Milliseconds to way until StartTransaction is allowed by the server
  - in the current implementation, there should be a single transaction at once on the server side:
    this is the time to try before reporting an ESQldbRemote exception failure

**TSQLDBProxyConnection = class(TSQLDBConnection)**

*Implements an abstract proxy-like virtual connection to a DB engine*
- can be used e.g. for remote access or execution in a background thread

**constructor Create(aProperties: TSQLDBConnectionProperties); override;**
- Connect to a specified database engine

**function IsConnected: boolean; override;**
- Return TRUE if Connect has been already successfully called
function NewStatement: TSQLDBStatement; override;
   Initialize a new SQL query statement for the given connection

procedure Commit; override;
   Commit changes of a Transaction for this connection

procedure Connect; override;
   Connect to the specified database

procedure Disconnect; override;
   Stop connection to the specified database

procedure Rollback; override;
   Discard changes of a Transaction for this connection

procedure StartTransaction; override;
   Begin a Transaction for this connection

TSQldbProxyStatementAbstract = class(TSQLDBStatementWithParamsAndColumns)
   Implements a proxy-like virtual connection statement to a DB engine
   - abstract class, with no corresponding kind of connection, but allowing access to the mapped data
     via Column*() methods
   - will handle an internal binary buffer when the statement returned rows data, as generated by
     TSQLDBStatement.FetchAllToBinary()

function ColumnBlob(Col: integer): RawByteString; override;
   Return a Column as a blob value of the current Row, first Col is 0

function ColumnCurrency(Col: integer): currency; override;
   Return a Column currency value of the current Row, first Col is 0
   - should retrieve directly the 64 bit Currency content, to avoid any rounding/conversion error
     from floating-point types

function ColumnData(Col: integer): pointer;
   Direct access to the data buffer of the current row
   - points to Double/Currency value, or variable-length Int64/UTF8/Blob
   - points to nil if the column value is NULL

function ColumnDateTime(Col: integer): TDateTime; override;
   Return a Column floating point value of the current Row, first Col is 0

function ColumnDouble(Col: integer): double; override;
   Return a Column floating point value of the current Row, first Col is 0

function ColumnInt(Col: integer): Int64; override;
   Return a Column integer value of the current Row, first Col is 0

function ColumnNull(Col: integer): boolean; override;
   Returns TRUE if the column contains NULL

function ColumnString(Col: integer): string; override;
   Return a Column text value as generic VCL string of the current Row, first Col is 0
function ColumnType(Col: integer; FieldSize: PInteger=nil): TSQLDBFieldType; override;

The Column type of the current Row

function ColumnUTF8(Col: integer): RawUTF8; override;

Return a Column UTF-8 encoded text value of the current Row, first Col is 0

procedure ColumnsToBinary(W: TFileBufferWriter; Null: pointer; const ColTypes: TSQLDBFieldTypeDynArray); override;

Append current row content as binary stream
- will save one data row in optimized binary format (if not in Null)
- virtual method called by FetchAllToBinary()
- follows the format expected by TSQLDBProxyStatement

procedure ColumnsToJSON(WR: TJSONWriter); override;

Return all columns values into JSON content

property DataRowCount: integer read fDataRowCount;

Read-only access to the number of data rows stored

TSQLDBProxyStatement = class(TSQLDBProxyStatementAbstract)

Implements a proxy-like virtual connection statement to a DB engine
- is generated by TSQLDBProxyConnection kind of connection
- will use an internal binary buffer when the statement returned rows data, as generated by TSQLDBStatement.FetchAllToBinary() or JSON for ExecutePreparedAndFetchAllAsJSON() method (as expected by our ORM)

function FetchAllToBinary(Dest: TStream; MaxRowCount: cardinal=0; DataRowPosition: PCardinalDynArray=nil): cardinal; override;

Append all rows content as binary stream
- will save the column types and name, then every data row in optimized binary format (faster and smaller than JSON)
- you can specify a LIMIT for the data extent (default 0 meaning all data)
- generates the format expected by TSQLDBProxyStatement
- this overridden method will use the internal data copy of the binary buffer retrieved by ExecutePrepared, so would be almost immediate, and would allow e.g. direct consumption via our TSynSQLStatementDataSet
- note that DataRowPosition won't be set by this method: will be done e.g. in TSQLDBProxyStatementRandomAccess.Create

function Step(SeekFirst: boolean=false): boolean; override;

After a statement has been prepared via Prepare() + ExecutePrepared() or Execute(), this method must be called one or more times to evaluate it

function UpdateCount: integer; override;

Gets a number of updates made by latest executed statement
- this overridden method will return the integer value returned by cExecute command
procedure ExecutePrepared; override;

Execute a SQL statement
- for TSQLDBProxyStatement, preparation and execution are processed in one step, when this
  method is executed - as such, Prepare() won't call the remote process, but will just set fSQL
  - this overridden implementation will use out optimized binary format as generated by
    TSQLDBStatement.FetchAllToBinary(), and not JSON

procedure ExecutePreparedAndFetchAllAsJSON(Expanded: boolean; out JSON: RawUTF8); override;

Execute a prepared SQL statement and return all rows content as a JSON string
- JSON data is retrieved with UTF-8 encoding
  - if Expanded is true, JSON data is an array of objects, for direct use with any Ajax or .NET client:
    [ {"col1":val11,"col2":"val12"},{"col1":val121,... } ]
  - if Expanded is false, JSON data is serialized (used in TSQLTableJSON)
    { "FieldCount":1,"Values":["col1","col1","val11","val12",val121,... ] }
- BLOB field value is saved as Base64, in the "\uFFF0base64encodedbinary" format and contains
  true BLOB data
- this overridden implementation will use JSON for transmission, and binary encoding only for
  parameters (to avoid unneeded conversions, e.g. when called from mORMotDB.pas)

property ForceNoUpdateCount: boolean read fForceNoUpdateCount write fForceNoUpdateCount;

Force no UpdateCount method call on server side
- may be needed to reduce server load, if this information is not needed

TSQLDBRemoteConnectionPropertiesAbstract =
  class(TSQLDBProxyConnectionPropertiesAbstract)
  Client-side implementation of a remote connection to any SynDB engine
  - will compute binary compressed messages for the remote processing, ready to be served e.g.
    over HTTP via our SynDBRemote.pas unit
  - abstract class which should override its protected ProcessMessage() method e.g. by
    TSQLDBRemoteConnectionPropertiesTest or

TSQLDBRemoteConnectionPropertiesTest =
  class(TSQLDBRemoteConnectionPropertiesAbstract)
  Fake proxy class for testing the remote connection to any SynDB engine
  - resulting overhead due to our binary messaging: unnoticeable :)
  constructor Create(aProps: TSQLDBConnectionProperties; const aUserID,aPassword:
    RawUTF8; aProtocol: TSQLDBProxyConnectionProtocolClass); reintroduce;

Create a test redirection to an existing local connection property
- you can specify a User/Password credential pair to also test the authentication via
  TSynAuthentication
TSQLDBProxyStatementRandomAccess = class(TSQLDBProxyStatementAbstract)

Implements a virtual statement with direct data access
- is generated with no connection, but allows direct random access to any data row retrieved from TSQLDBStatement.FetchAllToBinary() binary data
- GotoRow() method allows direct access to a row data via Column*()
- is used e.g. by TSynSQLStatementDataSet of SynDBVCL unit

constructor Create(Data: PByte; DataLen: integer; DataRowPosition: PCardinalDynArray=nil; IgnoreColumnDataSize: boolean=false); reintroduce;

Initialize the internal structure from a given memory buffer
- by default, ColumnDataSize would be computed from the supplied data, unless you set IgnoreColumnDataSize=true to set the value to 0 (and force e.g. SynDBVCL TSynBinaryDataSet.InternalInitFieldDefs define the field as ftDefaultMemo)

function GotoRow(Index: integer; RaiseExceptionOnWrongIndex: Boolean=false): boolean;

Change the current data Row
- if Index<DataRowCount, returns TRUE and you can access to the data via regular Column*() methods
- can optionally raise an ESQLDBException if Index is not correct

function Step(SeekFirst: boolean=false): boolean; override;

Change cursor position to the next available row
- this unexpected overridden method will raise a ESQLDBException

procedure ExecutePrepared; override;

Execute a prepared SQL statement
- this unexpected overridden method will raise a ESQLDBException

TSQLDBLib = class(TObject)

Access to a native library
- this generic class is to be used for any native connection using an external library
- is used e.g. in SynDBOracle by TSQLDBOracleLib to access the OCI library, or by SynDBODBC to access the ODBC library

destructor Destroy; override;

Release associated memory and linked library

property Handle: HMODULE read fHandle write fHandle;
The associated library handle

property LibraryPath: TFileName read fLibraryPath;
The loaded library path

ESQLQueryException = class(ESynException)

Generic Exception type raised by the TQuery class
TQueryValue = object(TObject)

Pseudo-class handling a TQuery bound parameter or column value
- will mimic both TField and TParam classes as defined in standard DB unit, by pointing both
classes types to PQneryValue
- usage of an object instead of a class allow faster access via a dynamic array (and our
TDynArrayHashed wrapper) for fast property name handling (via name hashing) and pre-allocation
- it is based on an internal Variant to store the parameter or column value

procedure Clear;
  Set the column value to null

property AsBlob: TBlobData read GetBlob write SetBlob;
  Access the BLOB Value as an AnsiString
  - will work for all Delphi versions, including Unicode versions (i.e. since Delphi 2009)
  - for a BLOB parameter or column, you should use AsBlob or AsBlob properties instead of
    AsString (this later won’t work after Delphi 2007)

property AsBoolean: Boolean read GetBoolean write SetBoolean;
  Access the Value as boolean

property AsBytes: TBytes read GetAsBytes write SetAsBytes;
  Access the BLOB Value as array of byte (TBytes)
  - will work for all Delphi versions, including Unicode versions (i.e. since Delphi 2009)
  - for a BLOB parameter or column, you should use AsBlob or AsBlob properties instead of
    AsString (this later won’t work after Delphi 2007)

property AsCurrency: Currency read GetCurrency write SetCurrency;
  Access the Value as Currency
  - avoid any rounding conversion, as with AsFloat

property AsDate: TDateTime read GetDateTime write SetDateTime;
  Access the Value as TDate

property AsDateTime: TDateTime read GetDateTime write SetDateTime;
  Access the Value as TDateTime

property AsFloat: double read GetDouble write SetDouble;
  Access the Value as double

property AsInt64: Int64 read GetInt64 write SetInt64;
  Access the Value as Int64
  - note that under Delphi 5, Int64 is not handled: the Variant type only handle integer types, in
    this Delphi version :(

property AsInteger: integer read GetInteger write SetInteger;
  Access the Value as Integer

property AsLargeInt: Int64 read GetInt64 write SetInt64;
  Access the Value as Int64
  - note that under Delphi 5, Int64 is not handled: the Variant type only handle integer types, in
    this Delphi version :(
property AsString: string read GetString write SetString;
Access the Value as String
- used in the VCL world for both TEXT and BLOB content (BLOB content will only work in pre-Unicode Delphi version, i.e. before Delphi 2009)

property AsTime: TDateTime read GetDateTime write SetDateTime;
Access the Value as TTime

property AsVariant: Variant read GetVariant write SetVariant;
Access the Value as Variant

property AsWideString: SynUnicode read GetAsWideString write SetAsWideString;
Access the Value as an unicode String
- will return a WideString before Delphi 2009, and an UnicodeString for Unicode versions of the compiler (i.e. our SynUnicode type)

property Bound: Boolean write SetBound;
Just do nothing - here for compatibility reasons with Clear + Bound := true

property FieldName: string read fName;
The associated (field) name

property IsNull: Boolean read GetIsNull;
Returns TRUE if the stored Value is null

property Name: string read fName;
The associated (parameter) name

property ParamType: TParamType read fParamType write fParamType;
Parameter type for queries or stored procedures
TQuery = class(TObject)

Class mapping VCL DB TQuery for direct database process
- this class can mimic basic TQuery VCL methods, but won't need any BDE installed, and will be faster for field and parameters access than the standard TDataSet based implementation; in fact, OleDB replaces the BDE or the DBExpress layer, or access directly to the client library (e.g. for TSQLDBOracleConnectionProperties which calls oci.dll)
- it is able to run basic queries as such:
  
  Q := TQuery.Create(aSQLDBConnection);
  try
    Q.SQL.Clear; // optional
    Q.SQL.Add('select * from DOMAIN.TABLE');
    Q.SQL.Add('WHERE ID_DETAIL=:detail;');
    Q.ParamByName('DETAIL').AsString := '12342002010000430015';
    Q.Open;
    Q.First; // optional
    while not Q.Eof do begin
      assert(Q.FieldByName('id_detail').AsString='12342002010000430015');
      Q.Next;
    end;
  finally
    Q.Close; // optional
    Q.Free;
  end;

- since there is no underlying TDataSet, you can't have read and write access, or use the visual DB components of the VCL: it's limited to direct emulation of low-level SQL as in the above code, with one-direction retrieval (e.g. the Edit, Post, Append, Cancel, Prior, Locate, Lookup methods do not exist within this class)
- use ToDataSet() function from SynDBVCL.pas to create a TDataSet from such a TQuery instance, and link this request to visual DB components
- this class is Unicode-ready even before Delphi 2009 (via the TQueryValue AsWideString method), will natively handle Int64/TBytes field or parameter data, and will have less overhead than the standard DB components of the VCL
- you should better use TSQLDBStatement instead of this wrapper, but having such code-compatible TQuery replacement could make easier some existing code upgrade (e.g. to avoid deploying the deprecated BDE, generate smaller executable, access any database without paying a big fee, avoid rewriting a lot of existing code lines of a big application...)

constructor Create(aConnection: TSQLDBConnection);
  Initialize a query for the associated database connection

destructor Destroy; override;
  Release internal memory and statements

function ExecSQLAndReturnUpdateCount: integer;
  Begin the SQL query, for a non SELECT statement
  - will parse the entered SQL statement, and bind parameters
  - the query will be released with a call to Close within this method
  - this method will return the number of updated rows (i.e. PreparedSQLDBStatement.UpdateCount)
function FieldByName(const aFieldName: string): TField;
    Retrieve a column value from the current opened SQL query row
    - will raise an ESQLQueryException error in case of error, e.g. if no column name matches the supplied name

function FindField(const aFieldName: string): TField;
    Retrieve a column value from the current opened SQL query row
    - will return nil in case of error, e.g. if no column name matches the supplied name

function ParamByName(const aParamName: string; CreateIfNotExisting: boolean=true): TParam;
    Access a SQL statement parameter, entered as :aParamName in the SQL
    - if the requested parameter do not exist yet in the internal fParams list, AND if CreateIfNotExisting=true, a new TQueryValue instance will be created and registered

procedure Close;
    End the SQL query
    - will release the SQL statement, results and bound parameters
    - the query should be released with a call to Close before reopen

procedure ExecSQL;
    Begin the SQL query, for a non SELECT statement
    - will parse the entered SQL statement, and bind parameters
    - the query will be released with a call to Close within this method
    - will return the number of updated rows (i.e. PreparedSQLDBStatement.UpdateCount)

procedure First;
    After a successful Open, will get the first row of results

procedure Next;
    After successful Open and First, go the the next row of results

procedure Open;
    Begin the SQL query, for a SELECT statement
    - will parse the entered SQL statement, and bind parameters
    - will then execute the SELECT statement, ready to use First/Eof/Next methods, the returned rows being available via FieldByName methods

procedure Prepare;
    A do-nothing method, just available for compatibility purpose

property Active: Boolean read GetActive;
    Equals true if the query is opened

property Bof: Boolean read GetBof;
    Equals true if on first row

property Connection: TSQLDBConnection read fConnection;
    The associated database connection

property Eof: Boolean read GetEof;
    Equals true if there is some rows pending
property FieldCount: integer read GetFieldCount;
  The number of columns in the current opened SQL query row

property Fields[aIndex: integer]: TField read GetField;
  Retrieve a column value from the current opened SQL query row
  - will return nil in case of error, e.g. out of range index

property IsEmpty: Boolean read GetIsEmpty;
  Equals true if there is no row returned

property ParamCount: integer read GetParamCount;
  The number of bound parameters in the current SQL statement

property Params[aIndex: integer]: TParam read GetParam;
  Retrieve a bound parameters in the current SQL statement
  - will return nil in case of error, e.g. out of range index

property PreparedSQLDBStatement: ISQLDBStatement read fPrepared;
  Non VCL property to access the internal SynDB prepared statement
  - is nil if the TQuery is not prepared (e.g. after Close)

property RecordCount: integer read GetRecordCount;
  Returns 0 if no record was retrieved, 1 if there was some records
  - not the exact count: just here for compatibility purpose with code like if aQuery.RecordCount>0 then ...

property SQL: TStringList read fSQL;
  The SQL statement to be executed
  - statement will be prepared and executed via Open or ExecSQL methods
  - SQL.Clear will force a call to the Close method (i.e. reset the query, just as with the default VCL implementation)

property SQLAsText: string read GetSQLAsText;
  The SQL statement with inlined bound parameters

property Tag: PtrInt read fTag write fTag;
  User-customizable number attached to this instance
  - for compatibility with TComponent

Types implemented in the SynDB unit

PQueryValue = ^TQueryValue;
  Pointer to TQuery bound parameter or column value

TBlobData = RawByteString;
  Generic type used by TQuery / TQueryValue for BLOBs fields

TField = PQueryValue;
  Pointer mapping the VCL DB TField class
  - to be used e.g. with code using local TField instances in a loop

TOnBatchInsert = procedure(Props: TSQLDBConnectionProperties; const TableName: RawUTF8; const FieldNames: TRawUTF8DynArray; const FieldTypes: TSQLDBFieldTypeArray; RowCount: integer; const FieldValues: TRawUTF8DynArrayDynArray) of object;
  Defines a callback signature able to handle multiple INSERT
- may execute e.g. for 2 fields and 3 data rows on a database engine implementing INSERT with multiple VALUES (like MySQL, PostgreSQL, NexusDB, MSSQL or SQLite3), as implemented by TSQLDBConnectionProperties.MultipleValuesInsert():

```
INSERT INTO TableName(FieldNames[0],FieldNames[1]) VALUES
  (FieldValues[0][0],FieldValues[1][0]),
  (FieldValues[0][1],FieldValues[1][1]),
  (FieldValues[0][2],FieldValues[1][2]);
```

- for other kind of DB which do not support multi values INSERT, may execute a dedicated driver command, like MSSQL "bulk insert" or Firebird "execute block"

```
TOnSQLDBInfo = procedure(Sender: TSQLDBStatement; const Msg: RawUTF8) of object;
  
  Event handler called when the low-level driver send some warning information
  - errors will trigger Exceptions, but sometimes the database driver returns some non critical information, which is logged and may be intercepted using the TSQLDBConnectionProperties.OnStatementInfo property
  - may be used e.g. to track ORA-28001 or ORA-28002 about account expire
  - is currently implemented by SynDBOracle, SynDBODBC and SynOleDb units

TOnSQLDBProcess = procedure(Sender: TSQLDBConnection; Event: TOnSQLDBProcessEvent) of object;
  
  Event handler called during all external DB process
  - event handler is specified by TSQLDBConnectionProperties.OnProcess or TSQLDBConnection.OnProperties properties

TOnSQLDBProcessEvent = ( speConnected, speDisconnected, speNonActive, speActive, speConnectionLost, speReconnected, speStartTransaction, speCommit, speRollback );
  
  Possible events notified to TOnSQLDBProcess callback method
  - event handler is specified by TSQLDBConnectionProperties.OnProcess or TSQLDBConnection.OnProcess properties
  - speConnected / speDisconnected will notify TSQLDBConnection.Connect and TSQLDBConnection.Disconnect calls
  - speNonActive / speActive will be used to notify external DB blocking access, so can be used e.g. to change the mouse cursor shape (this trigger is re-entrant, i.e. it will be executed only once in case of nested calls)
  - speReconnected will be called if TSQLDBConnection did successfully recover its database connection (on error, TQuery will call speConnectionLost): this event will be called by TSQLDBConnection.Connect after a regular speConnected notification
  - speConnectionLost will be called by TQuery in case of broken connection, and if Disconnect/Reconnect did not restore it as expected (i.e. speReconnected)
  - speStartTransaction / speCommit / speRollback will notify the corresponding TSQLDBConnection.StartTransaction, TSQLDBConnection.Commit and TSQLDBConnection.Rollback methods

TParam = PQueryValue;
  
  Pointer mapping the VCL DB TParam class
  - to be used e.g. with code using local TParam instances

TParamType = ( ptUnknown, ptInput, ptOutput, ptInputOutput, ptResult );
  
  Represent the use of parameters on queries or stored procedures
  - same enumeration as with the standard DB unit from VCL

TQueryValueDynArray = array of TQueryValue;
  
  A dynamic array of TQuery bound parameters or column values
- TQuery will use TDynArrayHashed for fast search

TSQLDBColumnCreateDynArray = array of TSQLDBColumnCreate;
Used to define how a table is to be created

TSQLDBColumnDefineDynArray = array of TSQLDBColumnDefine;
Used to define the column layout of a table schema
- e.g. for TSQLDBConnectionProperties.GetFields

TSQLDBColumnPropertyDynArray = array of TSQLDBColumnProperty;
Used to define a table/field column layout

TSQLDBConnectionPropertiesClass = class of TSQLDBConnectionProperties;
Specify the class of TSQLDBConnectionProperties
- sometimes used to create connection properties instances, from a set of available classes (see e.g. SynDBExplorer or sample 16)

TSQLDBConnectionPropertiesThreadSafeThreadingMode = ( tmThreadPool, tmMainConnection, tmBackgroundThread );
Threading modes set to TSQLDBConnectionPropertiesThreadSafe.ThreadingMode
- default mode is to use a Thread Pool, i.e. one connection per thread
- or you can force to use the main connection
- or you can use a shared background thread process (not implemented yet)
- last two modes could be used for embedded databases (SQLite3/FireBird), when multiple connections may break stability, consume too much resources and/or decrease performance

TSQLDBDefinition = ( dUnknown, dDefault, dOracle, dMSSQL, dJet, dMySQL, dSQLite, dFirebird, dNexsusDB, dPostgreSQL, dB2, dInformix );
The known database definitions
- will be used e.g. for TSQLDBConnectionProperties.SQLFieldCreate(), or for OleDB/ODBC/ZDBC tuning according to the connected database engine

TSQLDBDefinitionLimitPosition = ( posNone, posWhere, posSelect, posAfter, posOuter );
Where the LIMIT clause should be inserted for a given SQL syntax
- used by TSQLDBDefinitionLimitClause and SQLLimitClause() method

TSQLDBDefinitions = set of TSQLDBDefinition;
Set of the available database definitions

TSQLDBFieldTypeDefinition = array[TSQLDBFieldType] of RawUTF8;
An array of RawUTF8, for each existing column type
- used e.g. by SQLCreate method
- ftUnknown maps int32 field (e.g. boolean), ftNull maps RawUTF8 index # field, ftUTF8 maps RawUTF8 blob field, other types map their default kind
- for UTF-8 text, ftUTF8 will define the BLOB field, whereas ftNull will expect to be formatted with an expected field length in ColumnAttr
- the RowID definition will expect the ORM to create an unique identifier, and will use the ftInt64 type definition for this and send it with the INSERT statement (some databases, like Oracle, do not support standard's IDENTITY attribute) - see http://troels.arvin.dk/db/rdbms

TSQLDBIndexDefineDynArray = array of TSQLDBIndexDefine;
Used to describe extended index definition of a table schema
- e.g. for TSQLDBConnectionProperties.GetIndexes

TSQLDBParamDynArray = array of TSQLDBParam;
Dynamic array used to store standard binding parameters
- used e.g. by TSQLDBStatementWithParams (and its inherited TSQLDBOracleStatement)

```
TSQDBParamInOutType = ( paramIn, paramOut, paramInOut );
```

The diverse type of bound parameters during a statement execution
- will be paramIn by default, which is the case 90% of time
- could be set to paramOut or paramInOut if must be refreshed after execution (for calling a stored procedure expecting such parameters)

```
TSQDBProcColumnDefineDynArray = array of TSQDBProcColumnDefine;
```

Used to define the parameter/column layout of a stored procedure schema
- e.g. for TSQLDBConnectionProperties.GetProcedureParameters

```
TSQLDBProxyConnectionCommand = ( cGetToken, cGetDBMS, cConnect, cDisconnect, cTryStartTransaction, cCommit, cRollback, cServerTimestamp, cGetFields, cGetIndexes, cGetTableNames, cGetForeignKeys, cExecute, cExecuteToBinary, cExecuteToJSON, cExecuteToExpandedJSON, cQuit, cExceptionRaised );
```

Proxy commands implemented by TSQLDBProxyConnectionProperties.Process()
- method signature expect "const Input" and "var Output" arguments
- Input is not used for cConnect, cDisconnect, cGetForeignKeys, cTryStartTransaction, cCommit, cRollback and cServerTimestamp
- Input is the TSQLDBProxyConnectionProperties instance for cInitialize
- Input is the RawUTF8 table name for most cGet* metadata commands
- Input is the SQL statement and associated bound parameters for cExecute, cExecuteToBinary, cExecuteToJSON, and cExecuteToExpandedJSON, encoded as TSQLDBProxyConnectionCommandExecute record
- Output is not used for cConnect, cDisconnect, cCommit, cRollback and cExecute
- Output is not used for cConnect, cDisconnect, cCommit, cRollback and cExecute
- Output is TSQLDBDefinition (i.e. DBMS type) for cInitialize
- Output is TTimeLog for cServerTimestamp
- Output is boolean for cTryStartTransaction
- Output is TSQLDBColumnDefineDynArray for cGetFields
- Output is TSQLDBIndexDefineDynArray for cGetIndexes
- Output is TSynNameValue (fForeignKeys) for cGetForeignKeys
- Output is TRawUTF8DynArray for cGetTableNames
- Output is RawByteString result data for cExecuteToBinary
- Output is UpdateCount: integer text for cExecute
- Output is RawUTF8 result data for cExecuteToJSON and cExecuteToExpandedJSON
- calls could be declared as such:
  - Process(cGetToken,?,result: Int64);
  - Process(cGetDBMS,User#1Hash: RawUTF8,fDBMS: TSQLDBDefinition);
  - Process(cConnect,?,?);
  - Process(cDisconnect,?,?);
  - Process(cTryStartTransaction,?,started: boolean);
  - Process(cCommit,?,?,);
  - Process(cRollback,?,?,);
  - Process(cServerTimestamp,?,result: TTimeLog);
  - Process(cGetFields,aTableName: RawUTF8,Fields: TSQLDBColumnDefineDynArray);
  - Process(cGetIndexes,aTableName: RawUTF8,Indexes: TSQLDBIndexDefineDynArray);
  - Process(cGetTableNames,?,Tables: TRawUTF8DynArray);
  - Process(cGetForeignKeys,?,fForeignKeys: TSynNameValue);
  - Process(cExecute,Request: TSQLDBProxyConnectionCommandExecute,UpdateCount: integer);
  - Process(cExecuteToBinary,Request: TSQLDBProxyConnectionCommandExecute,Data: RawByteString);
  - Process(cExecuteToJSON,Request: TSQLDBProxyConnectionCommandExecute,JSON: RawUTF8);
  - Process(cExecuteToExpandedJSON,Request: TSQLDBProxyConnectionCommandExecute,JSON: RawUTF8);
  - cExceptionRaised is a pseudo-command, used only for sending an exception to the client in case of
The known column data types corresponding to our TSQLDBFieldType types
- will be used e.g. for TSQLDBConnectionProperties.FieldCreate()
- see TSQLDBFieldTypeDefinition documentation to find out the mapping

The known column data types corresponding to our TSQLDBFieldType types
- will be used e.g. for TSQLDBConnectionProperties.FieldCreate()
- SQLite3 doesn't expect any field length, neither PostgreSQL, so set to 0

The known database engines handling CREATE INDEX IF NOT EXISTS statement
The known database engines handling CREATE INDEX on BLOB columns
- SQLite3 does not have any issue about indexing any column
- PostgreSQL is able to index TEXT columns, which are some kind of CLOB

The known SQL statement to retrieve the server date and time
- The DESC clause shall be used for a CREATE INDEX statement
- only identified syntax exception is for FireBird

Return local server time by default Jet is local -> return local time SQLite is local -> return local time the known SQL syntax to limit the number of returned rows in a SELECT
- Position indicates if should be included within the WHERE clause, at the beginning of the SQL statement, or at the end of the SQL statement
- InsertFmt will replace '%' with the maximum number of lines to be retrieved
- used by TSQLDBConnectionProperties.AdaptSQLLimitForEngineList()

The SQL text corresponding to the identified WHERE operators for a SELECT

Conversion matrix from TSQLDBFieldType into variant type

Functions or procedures implemented in the **SynDB** unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundArrayToJSONArray</td>
<td>Create a JSON array from an array of UTF-8 bound values</td>
<td>1265</td>
</tr>
<tr>
<td>LogTruncatedColumn</td>
<td>FtUnknown, ftNull, ftInt64, ftDouble, ftCurrency, ftDate, ftUTF8, ftBlob function helper logging some column truncation information text</td>
<td>1265</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>ReplaceParamsByNames</td>
<td>Replace all '?' in the SQL statement with named parameters like :AA :AB..</td>
<td>1265</td>
</tr>
<tr>
<td>ReplaceParamsByNumber</td>
<td>s Replace all '?' in the SQL statement with indexed parameters like $1 $2...</td>
<td>1265</td>
</tr>
<tr>
<td>ToText</td>
<td>Retrieve the text of a given Database SQL dialect enumeration</td>
<td>1266</td>
</tr>
<tr>
<td>ToText</td>
<td>Retrieve the ready-to-be displayed text of proxy commands implemented by TSQLDBProxyConnectionProperties.Process()</td>
<td>1266</td>
</tr>
<tr>
<td>ToText</td>
<td>Retrieve the text of a given Database field type enumeration</td>
<td>1266</td>
</tr>
<tr>
<td>TrimLeftSchema</td>
<td>Retrieve a table name without any left schema</td>
<td>1266</td>
</tr>
<tr>
<td>TSQLDBFieldTypeToString</td>
<td>Retrieve the ready-to-be displayed text of a given Database field type enumeration</td>
<td>1266</td>
</tr>
</tbody>
</table>

**function** BoundArrayToJSONArray(const Values: TRawUTF8DynArray): RawUTF8;

Create a JSON array from an array of UTF-8 bound values
- as generated during array binding, i.e. with quoted strings 'one','t"wo' -> '{"one","t"wo}'
- and 1,2,3 -> '{1,2,3}'
- as used e.g. by PostgreSQL library

**procedure** LogTruncatedColumn(const Col: TSQLDBColumnProperty);

FtUnknown, ftNull, ftInt64, ftDouble, ftCurrency, ftDate, ftUTF8, ftBlob function helper logging some column truncation information text

**function** ReplaceParamsByNames(const aSQL: RawUTF8; var aNewSQL: RawUTF8; aStripSemicolon: boolean=true): integer;

Replace all '?' in the SQL statement with named parameters like :AA :AB..
- returns the number of ? parameters found within aSQL
- won't generate any SQL keyword parameters (e.g. :AS :OF :BY), to be compliant with Oracle OCI expectations
- any ending ';' character is deleted, unless aStripSemicolon is unset

**function** ReplaceParamsByNumbers(const aSQL: RawUTF8; var aNewSQL: RawUTF8; IndexChar: AnsiChar = '$'; AllowSemicolon: boolean = false): integer;

Replace all '?' in the SQL statement with indexed parameters like $1 $2...
- returns the number of ? parameters found within aSQL
- as used e.g. by PostgreSQL & Oracle (:1 :2) library
- if AllowSemicolon is false (by default), reject any statement with ; (Postgres do not allow ; inside prepared statement); it should be true for Oracle

**function** ToText(Field: TSQLDBFieldType): PShortString; overload;

Retrieve the text of a given Database field type enumeration
- see also TSQLDBFieldTypeToString() function
function ToText(cmd: TSQLDBProxyConnectionCommand): PShortString; overload;

Retrieve the ready-to-be displayed text of proxy commands implemented by TSQLDBProxyConnectionProperties.Process()

function ToText(DBMS: TSQLDBDefinition): PShortString; overload;

Retrieve the text of a given Database SQL dialect enumeration
- see also TSQLDBConnectionProperties.GetDBMSName() method

function TrimLeftSchema(const TableName: RawUTF8): RawUTF8;

Retrieve a table name without any left schema
- e.g. TrimLeftSchema('SCHEMA.TABLENAME')='TABLENAME'

function TSQLDBFieldTypeToString(aType: TSQLDBFieldType): TShort16;

Retrieve the ready-to-be displayed text of a given Database field type enumeration

Variables implemented in the SynDB unit

SynDBLog: TSynLogClass=TSynLog;

The TSynLog class used for logging for all our SynDB related units
- you may override it with TSQLLog, if available from mORMot.pas
- since not all exceptions are handled specifically by this unit, you may better use a common TSynLog class for the whole application or module
27.10. SynDBDataset.pas unit

Purpose: DB.pas TDataset-based direct access classes (abstract TQuery-like)
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the SynDBDataset unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td>SynVirtualDataSet</td>
<td>DB VCL read-only virtual dataset</td>
<td>1840</td>
</tr>
</tbody>
</table>

SynDBDataset class hierarchy

Objects implemented in the SynDBDataset unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESQLDBDataset</td>
<td>Exception type associated to generic TDataSet / DB.pas unit Dataset connection</td>
<td>1268</td>
</tr>
<tr>
<td>TSQLDBDatasetConnectionProperties</td>
<td>implement properties shared by via the DB.pas TQuery-like connections</td>
<td>1268</td>
</tr>
<tr>
<td>TSQLDBDatasetStatement</td>
<td>implements a statement via the DB.pas TDataSet/TQuery-like connection</td>
<td>1270</td>
</tr>
<tr>
<td>TSQLDBDatasetStatementAbstract</td>
<td>implements an abstract statement via the DB.pas TDataSet/TQuery-like connection</td>
<td>1268</td>
</tr>
</tbody>
</table>
ESQLDBDataset = class(ESQLDBException)

_exception type associated to generic TDataSet / DB.pas unit Dataset connection_

TSQLDBDatasetConnectionProperties = class(TSQLDBConnectionPropertiesThreadSafe)

_implement properties shared by via the DB.pas TQuery-like connections_

constructor Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8); override;

_Initialize the properties to connect via TDataSet database access_
- this overridden method will enable the BATCH process (emulated in TSQLDBStatement.ExecutePrepared, native e.g. for FireDAC)

property ForceInt64AsFloat: boolean read fForceInt64AsFloat write fForceInt64AsFloat;

_Set to true to force all Int64 content to be processed as a truncated float_
- by default, Int64 values will be bound either as an integer (if the value is within expected range), either as Int64 variant
- on some versions of Delphi, and some version of TDataSet (e.g. BDE), you may have to use a conversion to double to avoid a runtime error

property ForceUseWideString: boolean read fForceUseWideString write fForceUseWideString;

_Set to true to force all text content to be processed as WideString instead of the default faster AnsiString, for pre-Unicode version of Delphi_
- by default, UTF-8 text parameter or column will use an AnsiString value: for pre-Unicode Delphi, avoiding WideString/OleStr content will speed up the process a lot, if you are sure that the current charset matches the expected one (which is very likely)
- set this property to TRUE so that WideString will be used when working with the internal TDataSet, to avoid any character data loss: the access to the property will be slower, but you won't have any potential data loss
- if the text value contains only ASCII 7 bit characters, it won't be converted to WideString (since it is not necessary)
- starting with Delphi 2009, the TEXT content will be processed as an UnicodeString, so this property is not necessary for most cases, but it appeared that some providers expects it to be defined

TSQLDBDatasetStatementAbstract = class(TSQLDBStatementWithParamsAndColumns)

_implements an abstract statement via the DB.pas TDataSet/TQuery-like connection_
- dedicated abstract class, able to use any TDataSet with any kind of parameter linking (e.g. FireDAC/AnyDAC do have its own parameters type)

constructor Create(aConnection: TSQLDBConnection); override;

_Create a statement instance_

destructor Destroy; override;

_Release the prepared statement_

function ColumnBlob(Col: Integer): RawByteString; override;

_Return a Column as a blob value of the current Row, first Col is 0_
function ColumnCurrency(Col: Integer): currency; override;
  Return a Column currency value of the current Row, first Col is 0

function ColumnDateTime(Col: Integer): TDateTime; override;
  Return a Column date and time value of the current Row, first Col is 0

function ColumnDouble(Col: Integer): double; override;
  Return a Column floating point value of the current Row, first Col is 0

function ColumnInt(Col: Integer): Int64; override;
  Return a Column integer value of the current Row, first Col is 0

function ColumnNull(Col: Integer): boolean; override;
  Returns TRUE if the column contains NULL

function ColumnUTF8(Col: Integer): RawUTF8; override;
  Return a Column UTF-8 encoded text value of the current Row, first Col is 0

function Step(SeekFirst: boolean = false): boolean; override;
  Access the next or first row of data from the SQL Statement result
  - return true on success, with data ready to be retrieved by Column*() methods
  - return false if no more row is available (e.g. if the SQL statement is not a SELECT but an
    UPDATE or INSERT command)
  - if SeekFirst is TRUE, will put the cursor on the first row of results
  - raise an ESQLDBDataset on any error

procedure ColumnsToJSON(WR: TJSONWriter); override;
  Append all columns values of the current Row to a JSON stream
  - will use WR.Expand to guess the expected output format
  - BLOB field value is saved as Base64, in the "\uFFF0base64encodedbinary" format and contains
    true BLOB data

procedure ExecutePrepared; override;
  Execute a prepared SQL statement
  - parameters marked as ? should have been already bound with Bind*() functions
  - this implementation will also loop through all internal bound array of values (if any), to
    implement BATCH mode even if the database library does not support array binding (only
    SynDBFireDAC does support it yet)
  - this overridden method will log the SQL statement if sllSQL has been enabled in
    SynDBLog.Family.Level
  - raise an ESQLDBDataset on any error

procedure Prepare(const aSQL: RawUTF8; ExpectResults: boolean = false); overload;
  Prepare an UTF-8 encoded SQL statement
  - parameters marked as ? will be bound later, before ExecutePrepared call
  - if ExpectResults is TRUE, then Step() and Column*() methods are available to retrieve the data
    rows
  - raise an ESQLDBDataset on any error

procedure ReleaseRows; override;
  Close the associated TQuery when ISQLDBStatement is back in cache
procedure Reset; override;

Reset the previous prepared statement
- this overridden implementation will reset all bindings and the cursor state
- raise an ESQLDBDataset on any error

TSQLEBDBDatasetStatement = class(TSQLDEBDBDatasetStatementAbstract)

- implements a statement via the DB.pas TDataSet/TQuery-like connection
- you should not use this abstract class directly, but one inherited implementation with overridden
  Dataset*() protected methods to handle the internal fQuery: TDataSet property

procedure Prepare(const aSQL: RawUTF8; ExpectResults: boolean = false); overload;

Prepare an UTF-8 encoded SQL statement
- parameters marked as ? will be bound later, before ExecutePrepared call
- if ExpectResults is TRUE, then Step() and Column*() methods are available to retrieve the data
rows
- raise an ESQLDBDataset on any error
27.11. SynDBMidasVCL.pas unit

**Purpose**: Fill a VCL TClientDataset from SynDB data access
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynDBMidasVCL unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynDBVCL</td>
<td>DB VCL read/only dataset from SynDB data access</td>
<td>1304</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

**SynDBMidasVCL class hierarchy**

**Objects implemented in the SynDBMidasVCL unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSynDBDataSet</td>
<td>A TClientDataSet which allows to apply updates on a SynDB connection</td>
<td>1271</td>
</tr>
</tbody>
</table>

**TSynDBDataSet = class(TCustomClientDataSet)**

* A TClientDataSet which allows to apply updates on a SynDB connection
  - typical usage may be for instance over a SynDBRemote connection:
    ```pascal
    props := TSQLDBWinHTTPConnectionProperties.Create(....);
    ds := TSynDBDataSet.Create(MainForm);
    ds.CommandText := 'select * from people';
    ds.Open;
    // ... use ds as usual, including modifications
    ds.ApplyUpdates(0);
    ```

**constructor** Create(AOwner: TComponent); **override**

* Initialize the instance

**procedure** From(Statement: TSQLDBStatement; MaxRowCount: cardinal=0);

* Initialize the internal TDataSet from a SynDB TSQLDBStatement result set
  - the supplied TSQLDBStatement can then be freed by the caller, since a private binary copy will be owned by this instance (in fDataSet.Data)
property Connection: TSQLDBConnectionProperties read GetConnection write SetConnection;
    The associated SynDB connection

property DataSet: TSynDBSQLDataSet read fDataSet;
    The associated SynDB TDataSet, used to retrieve and update data

property IgnoreColumnDataSize: boolean read fIgnoreColumnDataSize write fIgnoreColumnDataSize;
    If field sizes should be left unset, allowing further filling with any data length
    - by default, ColumnDataSize would be computed from the supplied data, unless you set
    IgnoreColumnDataSize=true to set the value to 0 (and force e.g. SynDBVCL
    TSynBinaryDataSet_InternalInitFieldDefs define the field as ftDefaultMemo)

Types implemented in the SynDBMidasVCL unit

TClientDataSetMode = ( cdsNew, cdsAppend, cdsReplace );
    How ToClientDataSet functions will fill the TClientDataSet instance

Functions or procedures implemented in the SynDBMidasVCL unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToClientDataSet</td>
<td>Fetch a SynDB TQuery result set into a new VCL TClientDataSet</td>
<td>1273</td>
</tr>
<tr>
<td>ToClientDataSet</td>
<td>Fetch a SynDB TSQLDBStatement result set into a new VCL TClientDataSet</td>
<td>1273</td>
</tr>
<tr>
<td>ToClientDataSet</td>
<td>Fetch a SynDB ISQLDBRows result set into a new VCL TClientDataSet</td>
<td>1273</td>
</tr>
<tr>
<td>ToClientDataSet</td>
<td>Fetch a SynDB TQuery result into an existing VCL TClientDataSet</td>
<td>1273</td>
</tr>
<tr>
<td>ToClientDataSet</td>
<td>Fetch a SynDB TSQLDBStatement result into an existing VCL TClientDataSet</td>
<td>1273</td>
</tr>
</tbody>
</table>

function ToClientDataSet(aDataSet: TClientDataSet; aStatement: SynDB.TQuery; aMaxRowCount: integer=0; aMode: TClientDataSetMode=cdsReplace; aLogChange: boolean=false): boolean; overload;
    Fetch a SynDB TQuery result into an existing VCL TClientDataSet
    - if aMaxRowCount>0, will return up to the specified number of rows
    - current implementation will fill an existing TClientDataSet instance, from the supplied TQuery
    - for better speed with Delphi older than Delphi 2009 Update 3, it is recommended to use
      http://andy.jgknet.de/blog/bugfix-units/midas-speed-fix-12
function ToClientDataSet(aDataSet: TClientDataSet; aStatement: TSQLDBStatement; aMaxRowCount: integer=0; aMode: TClientDataSetMode=cdsReplace; aLogChange: boolean=false): boolean; overload;

Fetch a SynDB TSQLDBStatement result into an existing VCL TClientDataSet
- if aMaxRowCount>0, will return up to the specified number of rows
- current implementation will fill an existing TClientDataSet instance, from the supplied TSQLDBStatement content
- for better speed with Delphi older than Delphi 2009 Update 3, it is recommended to use http://andy.jgknet.de/blog/bugfix-units/midas-speed-fix-12

function ToClientDataSet(aOwner: TComponent; aStatement: ISQLDBRows; aMaxRowCount: integer=0): TSynDBDataSet; overload;

Fetch a SynDB ISQLDBRows result set into a new VCL TClientDataSet
- this overloaded function can use directly a result of the TSQLDBConnectionProperties.Execute() method, as such:
  ds1.DataSet := ToClientDataSet(self,props.Execute('select * from table',[]));

function ToClientDataSet(aOwner: TComponent; aStatement: TQuery; aMaxRowCount: integer=0): TSynDBDataSet; overload;

Fetch a SynDB TQuery result set into a new VCL TClientDataSet
- if aMaxRowCount>0, will return up to the specified number of rows
- current implementation will return a TClientDataSet instance, created from the supplied TQuery content
- for better speed with Delphi older than Delphi 2009 Update 3, it is recommended to use http://andy.jgknet.de/blog/bugfix-units/midas-speed-fix-12
- if you need a read/only TDataSet, you should better not use this function but ToDataSet() as defined in SynDBVCL which is much faster and uses much less resources

function ToClientDataSet(aOwner: TComponent; aStatement: TSQLDBStatement; aMaxRowCount: integer=0): TSynDBDataSet; overload;

Fetch a SynDB TSQLDBStatement result set into a new VCL TClientDataSet
- if aMaxRowCount=0, will return up to the specified number of rows
- current implementation will return a TClientDataSet instance, created from the supplied TSQLDBStatement content
- for better speed with Delphi older than Delphi 2009 Update 3, it is recommended to use http://andy.jgknet.de/blog/bugfix-units/midas-speed-fix-12
- if you need a read/only TDataSet, you should better not use this function but ToDataSet() function as defined in SynDBVCL which is much faster and uses much less resources
27.12. SynDBODBC.pas unit

**Purpose**: ODBC 3.x library direct access classes to be used with our SynDB architecture
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

### Units used in the SynDBODBC unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/cache/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
</tbody>
</table>

### Objects implemented in the SynDBODBC unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EODBCException</td>
<td>Generic Exception type, generated for ODBC connection</td>
<td>1275</td>
</tr>
<tr>
<td>TODBConnection</td>
<td>Implements a direct connection to the ODBC library</td>
<td>1276</td>
</tr>
<tr>
<td>TODBConnectionProperties</td>
<td>Will implement properties shared by the ODBC library</td>
<td>1275</td>
</tr>
<tr>
<td>TODBStatement</td>
<td>Implements a statement using a ODBC connection</td>
<td>1277</td>
</tr>
</tbody>
</table>
EODBException = class (ESQLDBException)
Generic Exception type, generated for ODBC connection

TODBCConnectionProperties = class (TSQLDBConnectionPropertiesThreadSafe)
Will implement properties shared by the ODBC library

constructor Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8); override;

Initialize the connection properties
- will raise an exception if the ODBC library is not available
- SQLConnect() API will be used if aServerName is set: it should contain the ODBC Data source name as defined in "ODBC Data Source Administrator" tool (C:\Windows\SysWOW64\odbcad32.exe for 32bit app on Win64) - in this case, aDatabaseName will be ignored
- SQLDriverConnect() API will be used if aServerName is "" and aDatabaseName is set - in this case, aDatabaseName should contain a full connection string like (e.g. for a local SQLEXPRESS instance):
  'DRIVER=SQL Server Native Client
  10.0;UID=.;server=./SQLEXPRESS;'+
  'Trusted_Connection=Yes;MARS_Connection=yes'  
  see @http://msdn.microsoft.com/en-us/library/ms715433 or when using Firebird ODBC:
  'DRIVER=Firebird/InterBase(r) driver;CHARSET=UTF8;UID=SYSDBA;PWD=masterkey;'
  'DBNAME=MyServer/3051:C:\database\myData.fdb'
  'CLIENT=fbembed.dll'
for IBM DB2 and its official driver:
  'Driver=IBM DB2 ODBC DRIVER;Database=SAMPLE;'+
  'Hostname=localhost;Port=50000;UID=db2admin;Pwd=db2Password'
  'Driver=PostgreSQL Unicode;Database=postgres;'+
  'Server=localhost;Port=5432;UID=postgres;Pwd=postgresPassword'
for MySQL - driver from https://dev.mysql.com/downloads/connector/odbc (note: 5.2.6 and 5.3.1 driver seems to be slow in ODBC.FreeHandle)
  'Driver=MySQL ODBC 5.2 UNICODE Driver;Database=test;'+
  'Server=localhost;Port=3306;UID=root;Pwd= '
for IBM Informix and its official driver:
  'Driver=IBM INFORMIX ODBC DRIVER;Database=SAMPLE;'+
  'Host=localhost;Server=<instance name on host>;Service=<service name in ../drivers/etc/services>;Protocol=olsoctcp;UID=<Windows/Linux user account>;
  Pwd=<Windows/Linux user account password>'

function NewConnection: TSQLDBConnection; override;
Create a new connection
- call this method if the shared MainConnection is not enough (e.g. for multi-thread access)
- the caller is responsible of freeing this instance
- this overridden method will create an TODBCConnection instance
procedure GetFields(const aTableName: RawUTF8; out Fields: TSQLDBColumnDefineDynArray); override;

Retrieve the column/field layout of a specified table
- will also check if the columns are indexed
- will retrieve the corresponding metadata from ODBC library if SQL direct access was not defined (e.g. for dDB2)

procedure GetForeignKeys; override;

Initialize fForeignKeys content with all foreign keys of this DB
- used by GetForeignKey method

procedure GetProcedureNames(out Procedures: TRawUTF8DynArray); override;

Retrieve a list of stored procedure names from current connection

procedure GetProcedureParameters(const aProcName: RawUTF8; out Parameters: TSQLDBProcColumnDefineDynArray); override;

Retrieve procedure input/output parameter information
- aProcName: stored procedure name to retrieve parameter information.
- Parameters: parameter list info (name, datatype, direction, default)

procedure GetTableNames(out Tables: TRawUTF8DynArray); override;

Get all table names
- will retrieve the corresponding metadata from ODBC library if SQL direct access was not defined

procedure GetViewNames(out Views: TRawUTF8DynArray); override;

Get all view names
- will retrieve the corresponding metadata from ODBC library if SQL direct access was not defined

property SQLDriverConnectPrompt: boolean read fSQLDriverConnectPrompt write fSQLDriverConnectPrompt;

If full connection string may prompt the user for additional information
- property used only with SQLDriverConnect() API (i.e. when aServerName is " and aDatabaseName contains a full connection string)
- set to TRUE to allow UI prompt if needed

TODBCConnection = class(TSQLDBConnectionThreadSafe)

Implements a direct connection to the ODBC library

constructor Create(aProperties: TSQLDBConnectionProperties); override;

Connect to a specified ODBC database

destructor Destroy; override;

Release memory and connection

function IsConnected: boolean; override;

Return TRUE if Connect has been already successfully called

function NewStatement: TSQLDBStatement; override;

Initialize a new SQL query statement for the given connection
- the caller should free the instance after use
procedure Commit; override;

  Commit changes of a Transaction for this connection
  - StartTransaction method must have been called before

procedure Connect; override;

  Connect to the ODBC library, i.e. create the DB instance
  - should raise an Exception on error

procedure Disconnect; override;

  Stop connection to the ODBC library, i.e. release the DB instance
  - should raise an Exception on error

procedure Rollback; override;

  Discard changes of a Transaction for this connection
  - StartTransaction method must have been called before

procedure StartTransaction; override;

  Begin a Transaction for this connection
  - current implementation do not support nested transaction with those methods: exception will
    be raised in such case

property DBMS: TSQLDBDefinition read fDBMS;
  The remote DBMS type, as retrieved at ODBC connection opening

property DBMSName: RawUTF8 read fDBMSName;
  The remote DBMS name, as retrieved at ODBC connection opening

property DBMSVersion: RawUTF8 read fDBMSVersion;
  The remote DBMS version, as retrieved at ODBC connection opening

property DriverName: RawUTF8 read fDriverName;
  The local driver name, as retrieved at ODBC connection opening

property SQLDriverFullString: RawUTF8 read fSQLDriverFullString;
  The full connection string (expanded from ServerName)

TODBCStatement = class(TSQLDBStatementWithParamsAndColumns)
  Implements a statement using a ODBC connection

constructor Create(aConnection: TSQLDBConnection); override;

  Create a ODBC statement instance, from an existing ODBC connection
  - the Execute method can be called once per TODBCStatement instance, but you can use the
    Prepare once followed by several ExecutePrepared methods
  - if the supplied connection is not of TOleDBConnection type, will raise an exception

destructor Destroy; override;

  Release all associated memory and ODBC handles

function ColumnBlob(Col: integer): RawByteString; override;

  Return a Column as a blob value of the current Row, first Col is 0
  - ColumnBlob() will return the binary content of the field is was not ftBlob, e.g. a 8 bytes
    RawByteString for a vtInt64/vtDouble/vtDate/vtCurrency, or a direct mapping of the
    RawUnicode
function ColumnCurrency(Col: integer): currency; override;

Return a Column currency value of the current Row, first Col is 0
- should retrieve directly the 64 bit Currency content, to avoid any rounding/conversion error from floating-point types

function ColumnDateTime(Col: integer): TDateTime; override;

Return a Column floating point value of the current Row, first Col is 0

function ColumnDouble(Col: integer): double; override;

Return a Column floating point value of the current Row, first Col is 0

function ColumnInt(Col: integer): Int64; override;

Return a Column integer value of the current Row, first Col is 0

function ColumnNull(Col: integer): boolean; override;

Returns TRUE if the column contains NULL

function ColumnUTF8(Col: integer): RawUTF8; override;

Return a Column UTF-8 encoded text value of the current Row, first Col is 0

function Step(SeekFirst: boolean=false): boolean; override;

After a statement has been prepared via Prepare() + ExecutePrepared() or Execute(), this method must be called one or more times to evaluate it
- you shall call this method before calling any Column*() methods
- return TRUE on success, with data ready to be retrieved by Column*()
- return FALSE if no more row is available (e.g. if the SQL statement is not a SELECT but an UPDATE or INSERT command)
- access the first or next row of data from the SQL Statement result: if SeekFirst is TRUE, will put the cursor on the first row of results, otherwise, it will fetch one row of data, to be called within a loop
- raise an EODBCException or ESQLDBException exception on any error

function UpdateCount: integer; override;

Returns the number of rows updated by the execution of this statement

procedure ColumnsToJSON(WR: TJSONWriter); override;

Append all columns values of the current Row to a JSON stream
- will use WR.Expand to guess the expected output format
- fast overridden implementation with no temporary variable
- BLOB field value is saved as Base64, in the "\uFFF0base64encodedbinary" format and contains true BLOB data

procedure ExecutePrepared; override;

Execute a prepared SQL statement
- parameters marked as ? should have been already bound with Bind*() functions
- this overridden method will log the SQL statement if sllSQL has been enabled in SynDBLog.Family.Level
- raise an EODBCException or ESQLDBException on any error
procedure Prepare(const aSQL: RawUTF8; ExpectResults: Boolean=false); overload;

    Prepare an UTF-8 encoded SQL statement
    - parameters marked as ? will be bound later, before ExecutePrepared call
    - if ExpectResults is TRUE, then Step() and Column*() methods are available to retrieve the data rows
    - raise an EODBCException or ESQLDBException on any error

procedure ReleaseRows; override;

    Close the ODBC statement cursor resources

procedure Reset; override;

    Reset the previous prepared statement
    - this overridden implementation will reset all bindings and the cursor state
    - raise an EODBCException on any error

Functions or procedures implemented in the SynDBODBC unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBCInstalledDriversList</td>
<td>List all ODBC drivers installed, by reading the Windows Registry</td>
<td>1279</td>
</tr>
</tbody>
</table>

function ODBCInstalledDriversList(const aIncludeVersion: Boolean; var aDrivers: TStrings): boolean;

    List all ODBC drivers installed, by reading the Windows Registry
    - aDrivers is the output driver list container, which should be either nil (to create a new TStringList), or any existing TStrings instance (may be from VCL
    - aIncludeVersion: include the DLL driver version as <driver name>=<dll version> in aDrivers (somewhat slower)
27.13. SynDBOracle.pas unit

*Purpose:* Oracle DB direct access classes (via OCI)
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

### Units used in the SynDBOracle unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
</tbody>
</table>

**SynDBOracle class hierarchy**

**Objects implemented in the SynDBOracle unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESQLDBOracle</td>
<td>Exception type associated to the native Oracle Client Interface (OCI)</td>
<td>1281</td>
</tr>
<tr>
<td>TOracleDate</td>
<td>Memory structure used to store a date and time in native Oracle format</td>
<td>1281</td>
</tr>
<tr>
<td>TSQLOracleConnection</td>
<td>Implements a direct connection to the native Oracle Client Interface (OCI)</td>
<td>1283</td>
</tr>
</tbody>
</table>
Synope mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLDBOracleConnectionProperties</td>
<td>Will implement properties shared by native Oracle Client Interface connections</td>
<td>1281</td>
</tr>
<tr>
<td>TSQLDBOracleStatement</td>
<td>Implements a statement via the native Oracle Client Interface (OCI)</td>
<td>1284</td>
</tr>
</tbody>
</table>

ESQLDBOracle = class(ESQLDBException)

Exception type associated to the native Oracle Client Interface (OCI)

TORoracleDate = object(TObject)
Memory structure used to store a date and time in native Oracle format
- follow the SQLT_DAT column type layout

function ToDateTime: TDateTime;
Convert an Oracle date and time into Delphi TDateTime
- this method will ignore any date before 30 Dec 1899 (i.e. any TDateTime result < 0), to avoid
  e.g. wrong DecodeTime() computation from retrieved value: if you need to retrieve dates before
  1899, you should better retrieve the content using ISO-8601 text encoding

function ToIso8601(Dest: PUTF8Char): integer; overload;
Convert an Oracle date and time into its textual expanded ISO-8601
- will fill up to 21 characters, including double quotes

procedure From(const aIso8601: RawUTF8); overload;
Convert textual ISO-8601 into native Oracle date and time format

procedure From(const aValue: TDateTime); overload;
Convert Delphi TDateTime into native Oracle date and time format

procedure From(aIso8601: PUTF8Char; Length: integer); overload;
Convert textual ISO-8601 into native Oracle date and time format

procedure ToIso8601(var aIso8601: RawByteString); overload;
Convert an Oracle date and time into its textual expanded ISO-8601
- return the ISO-8601 text, without double quotes

TSQLDBOracleConnectionProperties = class(TSQLDBConnectionPropertiesThreadSafe)
Will implement properties shared by native Oracle Client Interface connections
constructor Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8); override;

    Initialize the connection properties
    - we don't need a database name parameter for Oracle connection: only aServerName is to be set
    - you may specify the TNSName in aServerName, or a connection string like
      '/[host[:port]][/service_name]', e.g. '/sales-server:1523/sales'
    - connection is opened globally as UTF-8, to match the internal encoding of our units; but CHAR / NVARCHAR2 fields will use the Oracle charset as retrieved from the opened connection (to avoid any conversion error)

class function ExtractTnsName(const aServerName: RawUTF8): RawUTF8;
    Extract the TNS listener name from a Oracle full connection string
    - e.g. ExtractTnsName('1.2.3.4:1521/dbname') returns 'dbname'

function IsCachable(P:PUTF8Char): boolean; override;
    Determine if the SQL statement can be cached
    - always returns false, to force server-side caching only on this driver

function NewConnection: TSQLDBConnection; override;
    Create a new connection
    - call this method if the shared MainConnection is not enough (e.g. for multi-thread access)
    - the caller is responsible of freeing this instance
    - this overridden method will create an TSQLDBOracleConnection instance

property BlobPrefetchSize: integer read fBlobPrefetchSize write fBlobPrefetchSize;
    The size (in bytes) of LOB prefetch
    - is set to 4096 (4 KB) by default, but may be changed for tuned performance

property ClientVersion: RawUTF8 read GetClientVersion;
    Returns the Client version e.g. 'oci.dll rev. 11.2.0.1'

property EnvironmentInitializationMode: integer read fEnvironmentInitializationMode write fEnvironmentInitializationMode;
    The OCI initialization mode used for the connection
    - equals OCI_EVENTS or OCI_THREADED by default, since will likely be used in a multi-threaded context (even if this class is inheriting from TSQLDBConnectionPropertiesThreadSafe), and OCI_EVENTS is needed to support Oracle RAC Connection Load Balancing
    - can be tuned depending on the configuration or the Oracle version

property IgnoreORA01453OnStartTransaction: boolean read fIgnoreORA01453OnStartTransaction write fIgnoreORA01453OnStartTransaction;
    When we execute a SELECT statement across a database link, a transaction lock is placed on the undo segments (transaction is implicity started). Setting this options to true allow to ignore ORA-01453 during TSQLDBOracleConnection.StartTransaction call.
    - see Oracle documentation
    http://docs.oracle.com/cd/B28359_01/server.111/b28310/ds_appdev002.htm

property InternalBufferSize: integer read fInternalBufferSize write fInternalBufferSize;
    The size (in bytes) of the internal buffer used to retrieve rows in statements
    - default is 128 KB, which gives very good results
property OnPasswordChanged: TNotifyEvent read FOnPasswordChanged write FOnPasswordChanged;
  Password changed event

property OnPasswordExpired: TOnPasswordExpired read FOnPasswordExpired write FOnPasswordExpired;
  Password Expired event

property RowsPrefetchSize: integer read fRowsPrefetchSize write fRowsPrefetchSize;
  The size (in bytes) of rows data prefetch at OCI driver level
  - is set to 128 KB by default, but may be changed for tuned performance

property StatementCacheSize: integer read fStatementCacheSize write fStatementCacheSize;
  The number of prepared statements cached by OCI on the Client side
  - is set to 30 by default
  - only used if UseCache=true

property UseWallet: boolean read fUseWallet write fUseWallet;
  Use the Secure External Password Store for Password Credentials
  - see Oracle documentation
  http://docs.oracle.com/cd/B28359_01/network.111/b28531/authentication.htm#DBSEG97906

TSQLDBOracleConnection = class(TSQLDBConnectionThreadSafe)
  Implements a direct connection to the native Oracle Client Interface (OCI)

constructor Create(aProperties: TSQLDBConnectionProperties); override;
  Prepare a connection to a specified Oracle database server

destructor Destroy; override;
  Release memory and connection

function IsConnected: boolean; override;
  Return TRUE if Connect has been already successfully called

function NewStatement: TSQLDBStatement; override;
  Initialize a new SQL query statement for the given connection
  - if UseCache=true, this overridden implementation will use server-side Oracle statement cache
  - in this case, StatementCacheSize will define how many statements are to be cached
  - not that IsCachable() has been overridden to return false, so statement cache on client side is disabled
  - the caller should free the instance after use

function PasswordChange: Boolean;
  Allows to change the password of the current connected user
  - will first launch the OnPasswordExpired event to retrieve the new password, then change it
  and call OnPasswordChanged event on success

procedure Commit; override;
  Commit changes of a Transaction for this connection
  - StartTransaction method must have been called before
procedure Connect; override;

Connect to the specified Oracle database server
- should raise an Exception on error
- the connection will be globally opened with UTF-8 encoding; for CHAR / NVARCHAR2 fields, the DB charset encoding will be retrieved from the server, to avoid any truncation during data retrieval
- BlobPrefetchSize, RowsPrefetchSize and StatementCacheSize field values of the associated properties will be used to tune the opened connection

procedure Disconnect; override;

Stop connection to the specified Oracle database server
- should raise an Exception on error

procedure Rollback; override;

Discard changes of a Transaction for this connection
- StartTransaction method must have been called before

procedure StartTransaction; override;

Begin a Transaction for this connection
- current implementation do not support nested transaction with those methods: exception will be raised in such case
- by default, TSQLDBOracleStatement works in AutoCommit mode, unless StartTransaction is called

TSQLDBOracleStatement = class(TSQLDBStatementWithParamsAndColumns)

Implements a statement via the native Oracle Client Interface (OCI)
- those statements can be prepared on the Delphi side, but by default we enabled the OCI-side statement cache, not to reinvent the wheel this time
- note that bound OUT ftUTF8 parameters will need to be pre-allocated before calling - e.g. via BindTextU(StringOfChar(3000),paramOut)
- you can also bind an TInt64DynArray or TRawUTF8DynArray as parameter to be assigned later as an OCI_OBJECT so that you may write such statements:

var
  arr: TInt64DynArray = [1, 2, 3];
Query := TSQLDBOracleConnectionProperties.NewThreadSafeStatementPrepared('select * from table where table.id in '+'(select column_value from table(cast(? as SYS.ODCINUMBERLIST)))');
Query.BindArray(1,arr);
Query.ExecutePrepared;

( use SYS.ODCIVARCHAR2LIST type cast for TRawUTF8DynArray values)

constructor Create(aConnection: TSQLDBConnection); override;

Create an OCI statement instance, from an existing OCI connection
- the Execute method can be called once per TSQLDBOracleStatement instance, but you can use the Prepare once followed by several ExecutePrepared methods
- if the supplied connection is not of TOleDbConnection type, will raise an exception

constructor CreateFromExistingStatement(aConnection: TSQLDBConnection; aStatement: pointer);

Initialize the class from an existing OCI statement (and connection)
- to be called e.g. by ColumnCursor() for SQLT_RSET kind of column
Synopse mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

destructor Destroy; override;
Release all associated memory and OCI handles

function BoundCursor(Param: Integer): ISQLDBRows; override;
Return a special CURSOR parameter content as a SynDB result set
- this method is not about a column, but a parameter defined with BindCursor() before method execution
- Cursors are not handled internally by mORMot, but some databases (e.g. Oracle) usually use such structures to get data from stored procedures
- this method allow direct access to the data rows after execution
- this overridden method will allow direct access to the data rows

function ColumnBlob(Col: integer): RawByteString; override;
Return a Column as a blob value of the current Row, first Col is 0
- ColumnBlob() will return the binary content of the field is was not ftBlob, e.g. a 8 bytes RawByteString for a vtInt64/vtDouble/vtDate/vtCurrency, or a direct mapping of the RawUnicode

function ColumnBlobBytes(Col: integer): TBytes; override;
Return a Column as a blob value of the current Row, first Col is 0
- this function will return the BLOB content as a TBytes
- this default virtual method will call ColumnBlob()

function ColumnCurrency(Col: integer): currency; override;
Return a Column currency value of the current Row, first Col is 0
- should retrieve directly the 64 bit Currency content, to avoid any rounding/conversion error from floating-point types

function ColumnCursor(Col: integer): ISQLDBRows; override;
Return a special CURSOR Column content as a SynDB result set
- Cursors are not handled internally by mORMot, but Oracle usually use such structures to get data from stored procedures
- such columns are mapped as ftUTF8, with the rows converted to JSON
- this overridden method will allow direct access to the data rows

function ColumnDateTime(Col: integer): TDateTime; override;
Return a Column date and time value of the current Row, first Col is 0

function ColumnDouble(Col: integer): double; override;
Return a Column floating point value of the current Row, first Col is 0

function ColumnInt(Col: integer): Int64; override;
Return a Column integer value of the current Row, first Col is 0

function ColumnNull(Col: integer): boolean; override;
Returns TRUE if the column contains NULL
function ColumnToVariant(Col: integer; var Value: Variant): TSQLDBFieldType; override;

Return a Column as a variant
- this implementation will retrieve the data with no temporary variable (since TQuery calls this method a lot, we tried to optimize it)
- a ftUTF8 content will be mapped into a generic WideString variant for pre-Unicode version of Delphi, and a generic UnicodeString (=string) since Delphi 2009: you may not lose any data during charset conversion
- a ftBlob content will be mapped into a TBlobData AnsiString variant

function ColumnUTF8(Col: integer): RawUTF8; override;

Return a Column UTF-8 encoded text value of the current Row, first Col is 0

function Step(SeekFirst: boolean=false): boolean; override;

After a statement has been prepared via Prepare() + ExecutePrepared() or Execute(), this method must be called one or more times to evaluate it
- you shall call this method before calling any Column*() methods
- return TRUE on success, with data ready to be retrieved by Column*()
- return FALSE if no more row is available (e.g. if the SQL statement is not a SELECT but an UPDATE or INSERT command)
- access the first or next row of data from the SQL Statement result: if SeekFirst is TRUE, will put the cursor on the first row of results, otherwise, it will fetch one row of data, to be called within a loop
- raise an ESQLDBOracle on any error

function UpdateCount: integer; override;

Returns the number of rows updated by the execution of this statement

procedure BindCursor(Param: integer); override;

Bind a special CURSOR parameter to be returned as a SynDB result set
- Cursors are not handled internally by mORMot, but some databases (e.g. Oracle) usually use such structures to get data from stored procedures
- such parameters are mapped as ftUnknown, and is always of paramOut type
- use BoundCursor() method to retrieve the corresponding ISQLDBRows after execution of the statement
- this overridden method will prepare direct access to the data rows

procedure ColumnBlobFromStream(Col: integer; Stream: TStream); override;

Write a blob Column into the Stream parameter
- expected to be used with 'SELECT .. FOR UPDATE' locking statements

procedure ColumnBlobToStream(Col: integer; Stream: TStream); override;

Read a blob Column into the Stream parameter

procedure ColumnsToJSON(WR: TJSONWriter); override;

Append all columns values of the current Row to a JSON stream
- will use WR.Expand to guess the expected output format
- fast overridden implementation with no temporary variable (about 20% faster when run over high number of data rows)
- BLOB field value is saved as Base64, in the "\uFF0base64encodedbinary" format and contains true BLOB data
procedure ColumnToSQLVar(Col: Integer; var Value: TSQLVar; var Temp: RawByteString);

Return a Column as a TSQLVar value, first Col is 0
- the specified Temp variable will be used for temporary storage of svtUTF8/svtBlob values
- this implementation will retrieve the data with no temporary variable, and handling ftCurrency/NUMBER(22,0) as fast as possible, directly from the memory buffers returned by OCI:
  it will ensure best performance possible when called from TSQLVirtualTableCursorExternal.Column method as defined in mORMotDB unit (i.e. mORMot external DB access)

procedure ExecutePrepared; override;

Execute a prepared SQL statement
- parameters marked as ? should have been already bound with Bind*() functions
- raise an ESQLDBOracle on any error

procedure Prepare(const aSQL: RawUTF8; ExpectResults: Boolean=false); overload; override;

Prepare an UTF-8 encoded SQL statement
- parameters marked as ? will be bound later, before ExecutePrepared call
- if ExpectResults is TRUE, then Step() and Column*() methods are available to retrieve the data rows
- raise an ESQLDBOracle on any error
- if aSQL requires a trailing ';', you should end it with ';;' e.g. for

```
DB.ExecuteNoResult('CREATE OR REPLACE FUNCTION ORA_POC(MAIN_TABLE IN VARCHAR2, REC_COUNT IN NUMBER, BATCH_SIZE IN NUMBER) RETURN VARCHAR2' +
  ' AS LANGUAGE JAVA' +
  ' NAME ''OraMain.selectTable(java.lang.String, int, int) return java.lang.String'';;',[]);
```

procedure ReleaseRows; override;

Finalize the OCI cursor resources - not implemented yet

Types implemented in the SynDBOracle unit

TOnPasswordExpired = function (Sender: TSQLDBConnection; var APassword: RawUTF8): Boolean of object;
Event triggered when an expired password is detected
- will allow to provide a new password

TOracleDateArray = array[0..(maxInt div sizeof(TOracleDate))-1] of TOracleDate;
Wrapper to an array of TOracleDate items

Constants implemented in the SynDBOracle unit

OCI_UTF8 = $367;
Defined here for overriding OCI_CHARSET_UTF8/OCI_CHARSET_WIN1252 if needed

Variables implemented in the SynDBOracle unit
OCI_CHARSET_UTF8: cardinal = OCI_AL32UTF8;

*The OCI charset used for UTF-8 encoding*
- OCI_UTF8 is a deprecated encoding, and OCI_AL32UTF8 should be preferred
- but you can fallback for OCI_UTF8 for compatibility purposes

OCI_CHARSET_WIN1252: cardinal = OCI_WE8MSWIN1252;

*The OCI charset used for WinAnsi encoding*

SynDBOracleBlobChunksCount: integer = 250;

*How many blob chunks should be handled at once*

SynDBOracleOCIpath: TFileName;

*Optional folder where the Oracle Client Library is to be searched*
- by default, the oci.dll library is searched in the system PATH, then in %ORACLE_HOME%\bin
- you can specify here a folder name in which the oci.dll is to be found
27.14. SynDBPostgres.pas unit

Purpose: PostgreSQL direct access classes for SynDB units (not DB.pas based)
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license - see LICENSE.md

Units used in the SynDBPostgres unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
<tr>
<td></td>
<td>- implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/cache/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

ESQLDBPostgres class hierarchy

Objects implemented in the SynDBPostgres unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESQLDBPostgres</td>
<td>Exception type associated to the native libpg Interface</td>
<td>1290</td>
</tr>
</tbody>
</table>
Implements a connection via the libpq access layer

Connection properties which will implement an internal Thread-Safe connection pool

Implements a statement via a Postgres database connection

ESQLDBPostgres = class (ESQLDBException)
Exception type associated to the native libpq Interface

TSQLDBPostgresConnectionProperties = class (TSQLDBConnectionPropertiesThreadSafe)
Connection properties which will implement an internal Thread-Safe connection pool

constructor Create(const aServerName, aDatabaseName, aUserID, aPassword: RawUTF8);
override;

Initialize the properties
- raise an exception in case libpq is not thead-safe
- aDatabaseName can be a Connection URI - see
https://www.postgresql.org/docs/current/libpq-connect.html#LIBPQ-CONNSTRING
- if aDatabaseName contains connection URI with password we recommend to repeat password
  in aPassword parameter to prevent logging it (see
  TSQLDBConnectionProperties.DatabaseNameSafe)
- better to use environment variables and postgres config file for connection parameters

function NewConnection: TSQLDBConnection; override;
Create a new connection
- caller is responsible of freeing this instance
- this overridden method will create an TSQLDBPostgresConnection instance

function Oid2FieldType(cOID: cardinal): TSQLDBFieldType;
Add or replace mapping of OID into TSQLDBFieldType
- in case mapping for OID is not defined, returns ftUTF8

procedure MapOid(cOid: cardinal; fieldType: TSQLDBFieldType);
Add new (or override existed) OID to Field_Type mapping

TSQLDBPostgresConnection = class (TSQLDBConnectionThreadSafe)
Implements a connection via the libpq access layer

function IsConnected: boolean; override;
Return TRUE if Connect has been already successfully called

function NewStatement: TSQLDBStatement; override;
Create a new statement instance

Objects | Description | Page |
---------|-------------|------|
TSQLDBPostgresConnection | Implements a connection via the libpq access layer | 1290 |
TSQLDBPostgresConnectionProperties | Connection properties which will implement an internal Thread-Safe connection pool | 1290 |
TSQLDBPostgresStatement | Implements a statement via a Postgres database connection | 1291 |
procedure Commit; override;
   Commit changes of a Transaction for this connection
   - StartTransaction method must have been called before

procedure Connect; override;
   Connect to the specified server
   - should raise an ESQLDBPostgres on error

procedure Disconnect; override;
   Stop connection to the specified PostgreSQL database server
   - should raise an ESQLDBPostgres on error

procedure Rollback; override;
   Discard changes of a Transaction for this connection
   - StartTransaction method must have been called before

procedure StartTransaction; override;
   Begin a Transaction for this connection

property Direct: pointer read fPGConn;
   Direct access to the associated PPGconn connection

property PreparedCount: integer read fPreparedCount;
   How many prepared statements are currently cached for this connection

TSQLDBPostgresStatement = class(TSQLDBStatementWithParamsAndColumns)
   Implements a statement via a Postgres database connection

destructor Destroy; override;
   Finalize the statement for a given connection

function ColumnBlob(Col: integer): RawByteString; override;
   Return a Column as a blob value of the current Row, first Col is 0

function ColumnCurrency(Col: integer): currency; override;
   Return a Column currency value of the current Row, first Col is 0

function ColumnDateTime(Col: integer): TDateTime; override;
   Return a Column date and time value of the current Row, first Col is 0

function ColumnDouble(Col: integer): double; override;
   Return a Column floating point value of the current Row, first Col is 0

function ColumnInt(Col: integer): int64; override;
   Return a Column integer value of the current Row, first Col is 0

function ColumnNull(Col: integer): boolean; override;
   Returns TRUE if the column contains NULL

function ColumnUTF8(Col: integer): RawUTF8; override;
   Return a Column UTF-8 encoded text value of the current Row, first Col is 0
function Step(SeekFirst: boolean = False): boolean; override;

Access the next or first row of data from the SQL Statement result
- return true on success, with data ready to be retrieved by Column*() methods
- return false if no more row is available (e.g. if the SQL statement is not a SELECT but an
UPDATE or INSERT command)
- if SeekFirst is TRUE, will put the cursor on the first row of results
- raise an ESQLDBPostgres on any error

function UpdateCount: integer; override;

Gets a number of updates made by latest executed statement

procedure ColumnsToJSON(WR: TJSONWriter); override;

Append all columns values of the current Row to a JSON stream
- overriden method to avoid temporary memory allocation or conversion

procedure ExecutePrepared; override;

Execute a prepared SQL statement
- parameters marked as ? should have been already bound with Bind*() functions
- this implementation will also handle bound array of values (if any)
- this overridden method will log the SQL statement if sllSQL has been enabled in
SynDBLog.Family.Level
- raise an ESQLDBPostgres on any error

procedure Prepare(const aSQL: RawUTF8; ExpectResults: boolean = False); overload; override;

Prepare an UTF-8 encoded SQL statement
- parameters marked as ? will be bound later, before ExecutePrepared call
- if ExpectResults is TRUE, then Step() and Column*() methods are available to retrieve the data rows
- raise an ESQLDBPostgres on any error

procedure ReleaseRows; override;

Clear(fRes) when ISQLDBStatement is back in cache

procedure Reset; override;

Reset the previous prepared statement
- this overridden implementation will reset all bindings and the cursor state
- raise an ESQLDBPostgres on any error

property PreparedParamsCount: integer read fPreparedParamsCount;

How many parameters founded during prepare stage

Variables implemented in the SynDBPostgres unit

SynDBPostgresLibrary: TFileName;

Allow to specify a libpq library file name to use
27.15. SynDBRemote.pas unit

*Purpose:* Remote access to any RDBMS via HTTP using our SynDB architecture
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the *SynDBRemote* unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
<tr>
<td>SynCrtSock</td>
<td>Classes implementing TCP/UDP/HTTP client and server protocol - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1083</td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes - this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1204</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features - as a complement to SynCommons, which tended to increase too much - licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1721</td>
</tr>
</tbody>
</table>

**SynDBRemote class hierarchy**

Objects implemented in the *SynDBRemote* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLDBHTTPConnectionPropertiesAbstract</td>
<td>Implements a generic HTTP client, able to access remotely any SynDB</td>
<td>1295</td>
</tr>
<tr>
<td>TSQLDBHttpRequestConnectionProperties</td>
<td>Implements an abstract HTTP client via THttpRequest abstract class, able to access remotely any SynDB</td>
<td>1296</td>
</tr>
<tr>
<td>TSQLDBServerAbstract</td>
<td>Implements a generic HTTP server, able to publish any SynDB connection</td>
<td>1294</td>
</tr>
<tr>
<td>TSQLDBServerHttpApi</td>
<td>Implements a SynDB HTTP server using fast http.sys kernel-mode server</td>
<td>1295</td>
</tr>
<tr>
<td>TSQLDBServerSockets</td>
<td>Implements a SynDB HTTP server via the user-land Sockets API</td>
<td>1295</td>
</tr>
<tr>
<td>TSQLDBSocketConnectionProperties</td>
<td>Implements a HTTP client via sockets, able to access remotely any SynDB</td>
<td>1296</td>
</tr>
</tbody>
</table>
### Objects and Description

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLDBWinHTTPConnectionProperties</td>
<td>Implements a HTTP client via WinHTTP API, able to access remotely any SynDB</td>
<td>1296</td>
</tr>
<tr>
<td>TSQLDBWinINetConnectionProperties</td>
<td>Implements a HTTP client via WinINet API, able to access remotely any SynDB</td>
<td>1296</td>
</tr>
</tbody>
</table>

#### TSQLDBServerAbstract = class(TObject)

*Implements a generic HTTP server, able to publish any SynDB connection*

- do not instantiate this class, but rather use TSQLDBServerHttpApi or TSQLDBServerSockets - this abstract class won't set any HTTP server

```pascal
constructor Create(aProperties: TSQLDBConnectionProperties; const aDatabaseName: RawUTF8; const aPort: RawUTF8=SYNDB_DEFAULT_HTTP_PORT; const aUserName: RawUTF8=''; const aPassword: RawUTF8=''; aHttps: boolean=false; aThreadPoolCount: integer=1; aProtocol: TSQLDBProxyConnectionProtocolClass=nil; aThreadMode: TSQLDBConnectionPropertiesThreadSafeThreadingMode=tmMainConnection; aAuthenticate: TSynAuthenticationAbstract=nil); virtual;
```

*Publish the SynDB connection on a given HTTP port and URI*

- this generic constructor won't initialize the HTTP server itself: use overridden constructors instead
- URI would follow the supplied aDatabaseName parameter on the given port e.g. http://serverip:8092/remotedb for Create(aProps, 'remotedb');
- you can optionally register one user credential, or change the transmission Protocol which is TSQLDBRemoteConnectionProtocol by default
- aProperties.ThreadingMode will be set to the optional aThreadMode parameter tmMainConnection by default, which would also set ProcessLocked to TRUE - in fact, you should better use a single thread for the process, but you may define a small thread pool for the process IF the provider supports it

**Destructor** Destroy; **override;**

*Released used memory*

**Property** DatabaseName: RawUTF8 **read** fDatabaseName;

*The associated database name*

**Property** Port: RawUTF8 **read** fPort;

*The associated port number*

**Property** ProcessLocked: boolean **read** fProcessLocked **write** fProcessLocked;

*If the internal Process() method would be protected by a critical section*
- set to TRUE if constructor's aThreadMode is left to its default tmMainConnection value

**Property** Properties: TSQLDBConnectionProperties **read** fProperties **write** fProperties;

*The associated database connection properties*

**Property** Protocol: TSQLDBProxyConnectionProtocol **read** fProtocol **write** fProtocol;

*The associated communication protocol*
- to manage user authentication, use AuthenticateUser/DisauthenticateUser methods of Protocol.Authenticate
TSQLDBServerSockets = class(TSQLDBServerAbstract)

  Implements a SynDB HTTP server via the user-land Sockets API

  constructor Create(aProperties: TSQLDBConnectionProperties; const aDatabaseName: RawUTF8; const aPort: RawUTF8=SYNDB_DEFAULT_HTTP_PORT; const aUserName: RawUTF8=''; const aPassword: RawUTF8=''; aHttps: boolean=false; aThreadPoolCount: integer=1; aProtocol: TSQLDBProxyConnectionProtocolClass=nil; aThreadMode: TSQLDBConnectionPropertiesThreadSafeThreadingMode=tmMainConnection; aAuthenticate: TSynAuthenticationAbstract=nil); override;

  Publish the SynDB connection on a given HTTP port and URI using sockets
  - URI would follow the supplied aDatabaseName parameter on the given port e.g. http://serverip:8092/remotedb
  - you can optionally register one user credential
  - parameter aHttps is ignored by this class

TSQLDBServerHttpApi = class(TSQLDBServerAbstract)

  Implements a SynDB HTTP server using fast http.sys kernel-mode server
  - under Windows, this class is faster and more stable than TSQLDBServerSockets

  constructor Create(aProperties: TSQLDBConnectionProperties; const aDatabaseName: RawUTF8; const aPort: RawUTF8=SYNDB_DEFAULT_HTTP_PORT; const aUserName: RawUTF8=''; const aPassword: RawUTF8=''; aHttps: boolean=false; aThreadPoolCount: integer=1; aProtocol: TSQLDBProxyConnectionProtocolClass=nil; aThreadMode: TSQLDBConnectionPropertiesThreadSafeThreadingMode=tmMainConnection; aAuthenticate: TSynAuthenticationAbstract=nil); override;

  Publish the SynDB connection on a given HTTP port and URI using http.sys
  - URI would follow the supplied aDatabaseName parameter on the given port e.g. http://serverip:8092/remotedb
  - you can optionally register one user credential

TSQLDBHTTPConnectionPropertiesAbstract = class(TSQLDBRemoteConnectionPropertiesAbstract)

  Implements a generic HTTP client, able to access remotely any SynDB
  - do not instantiate this class, but rather use TSQLDBSocketConnectionProperties TSQLDBWinHTTPConnectionProperties

  property KeepAliveMS: cardinal read fKeepAliveMS write fKeepAliveMS;
  Time (in milliseconds) to keep the connection alive with the server
  - default is 60000, i.e. one minute

  property Port: RawByteString read GetPort;
  The associated port number

  property Server: RawByteString read GetServer;
  The associated server IP address or name
TSQLDBSocketConnectionProperties =
class(TSQLDBHTTPConnectionPropertiesAbstract)
$\text{Implements a HTTP client via sockets, able to access remotely any SynDB}$

constructor Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8);
override;
$\text{Initialize the properties for remote access via HTTP using sockets}$

- aServerName should be the HTTP server address as 'server:port'
- aDatabaseName would be used to compute the URI as in TSQLDBServerAbstract
- the user/password credential should match server-side authentication

destructor Destroy; override;
$\text{Released used memory}$

property Socket: THttpClientSocket read fSocket;
$\text{Low-level direct access to the Socket implementation instance}$

TSQLDBHTTPRequestConnectionProperties =
class(TSQLDBHTTPConnectionPropertiesAbstract)
$\text{Implements an abstract HTTP client via THttpRequest abstract class, able to access remotely any SynDB}$

- never instantiate this class, but rather TSQLDBWinHTTPConnectionProperties or TSQLDBWinInetConnectionProperties

destructor Destroy; override;
$\text{Released used memory}$

property Client: THttpRequest read fClient;
$\text{Low-level direct access to the WinHTTP implementation instance}$

TSQLDBWinHTTPConnectionProperties =
class(TSQLDBHTTPRequestConnectionProperties)
$\text{Implements a HTTP client via WinHTTP API, able to access remotely any SynDB}$

constructor Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8);
override;
$\text{Initialize the properties for remote access via HTTP using WinHTTP}$

- aServerName should be the HTTP server address as 'server:port'
- aDatabaseName would be used to compute the URI as in TSQLDBServerAbstract
- the user/password credential should match server-side authentication

TSQLDBWinInetConnectionProperties =
class(TSQLDBHTTPRequestConnectionProperties)
$\text{Implements a HTTP client via WinInet API, able to access remotely any SynDB}$
constructor Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8); override;

   *Initialize the properties for remote access via HTTP using WinINet*
   - aServerName should be the HTTP server address as 'server:port'
   - aDatabaseName would be used to compute the URI as in TSQLDBServerAbstract
   - the user/password credential should match server-side authentication

Types implemented in the SynDBRemote unit

TSQLDBServerClass = class of TSQLDBServerAbstract;
   *Used to define the HTTP server class for publishing a SynDB connection*

TSQLDBServerRemote = TSQLDBServerHttpApi;
   *The default SynDB HTTP server class on each platform*

Constants implemented in the SynDBRemote unit

SYNDB_DEFAULT_HTTP_PORT = '8092';
   *Default HTTP port to be used for SynDB remote access if none is specified*
27.16. *SynDBSQLite3.pas* unit

*Purpose:* SQLite3 direct access classes to be used with our SynDB architecture  
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The *SynDBSQLite3* unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.2.1</td>
<td>The SQLite3 engine shall be embedded to the framework</td>
<td>2548</td>
</tr>
</tbody>
</table>

Units used in the *SynDBSQLite3* unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| *SynCommons*| Common functions used by most Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                  | 717  |
| *SynDB*     | Abstract database direct access classes  
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                  | 1204 |
| *SynLog*    | Logging functions used by Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                  | 1363 |
| *SynSQLite3*| SQLite3 Database engine direct access  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                  | 1646 |
| *SynTable*  | Filter/database/cache/buffer/security/search/multithread/OS features  
- as a complement to SynCommons, which tended to increase too much  
- licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                  | 1721 |

![SynDBSQLite3 class hierarchy](image)

Objects implemented in the *SynDBSQLite3* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLDBSQLite3Connection</td>
<td>Implements a direct connection to the SQLite3 engine</td>
<td>1299</td>
</tr>
<tr>
<td>TSQLDBSQLite3ConnectionProperties</td>
<td>Will implement properties shared by the SQLite3 engine</td>
<td>1299</td>
</tr>
</tbody>
</table>
 Synopse mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

Objects | Description | Page
--- | --- | ---
TSQLDBSQLite3Statement | Implements a statement using the SQLite3 engine | 1300

**TSQLDBSQLite3ConnectionProperties = class(TSQLDBConnectionProperties)**

*Will implement properties shared by the SQLite3 engine*

*Used for DI-2.2.1 (page 2548)*.

**constructor** Create(aDB: TSQLDatabase); reintroduce; overload;

*Initialize access to an existing SQLite3 engine*
- this overloaded constructor allows to access via SynDB methods to an existing SQLite3 database, e.g. TSQLRestServerDB.DB (from mORMotSQLite3.pas)

**constructor** Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8); overload; override;

*Initialize access to a SQLite3 engine with some properties*
- only used parameter is aServerName, which should point to the SQLite3 database file to be opened (one will be created if none exists)
- if specified, the password will be used to cypher this file on disk (the main SQLite3 database file is encrypted, not the wal file during run); the password may be a JSON-serialized TSynSignerParams object, or will use AES-OFB-128 after SHAKE_128 with rounds=1000 and a fixed salt on plain password text
- other parameters (DataBaseName, UserID) are ignored

**function** NewConnection: TSQLDBConnection; override;

*Create a new connection*
- call this method if the shared MainConnection is not enough (e.g. for multi-thread access)
- the caller is responsible of freeing this instance

**property** MainSQLite3DB: TSQLDataBase read GetMainDB;

*Direct access to the main SQLite3 DB instance*
- can be used to tune directly the database properties

**property** UseMormotCollations: boolean read fUseMormotCollations write SetUseMormotCollations;

*TRUE if you want the SQL creation fields to use mORMot collation*
- default value is TRUE for use within the mORMot framework, to use dedicated UTF-8 collation and full Unicode support, and Iso8601 handling
- when set to FALSE, SQLCreate() method will return standard ASCII SQLite collations for TEXT: it will make interaction with other programs more compatible, at database file level

**TSQLDBSQLite3Connection = class(TSQLDBConnection)**

*Implements a direct connection to the SQLite3 engine*

*Used for DI-2.2.1 (page 2548)*.

**function** IsConnected: boolean; override;

*Return TRUE if Connect has been already successfully called*
function NewStatement: TSQLDBStatement; override;
  *Initialize a new SQL query statement for the given connection*
  - the caller should free the instance after use

procedure Commit; override;
  *Commit changes of a Transaction for this connection*
  - StartTransaction method must have been called before

procedure Connect; override;
  *Connect to the SQLite3 engine, i.e. create the DB instance*
  - should raise an Exception on error

procedure Disconnect; override;
  *Stop connection to the SQLite3 engine, i.e. release the DB instance*
  - should raise an Exception on error

procedure Rollback; override;
  *Discard changes of a Transaction for this connection*
  - StartTransaction method must have been called before

procedure StartTransaction; override;
  *Begin a Transaction for this connection*
  - current implementation do not support nested transaction with those methods: exception will be raised in such case

property DB: TSQLDataBase read fDB;
  *The associated SQLite3 DB instance*
  - assigned to not nil after successful connection

property LockingMode: TSQLLockingMode read GetLockingMode write SetLockingMode;
  *Query or change the SQLite3 file-based locking mode, i.e. the way it locks the file*
  - default lmNormal is ACID and safe
  - lmExclusive gives better performance in case of a number of write transactions, so can be used to release a mORMot server power: but you won't be able to access the database file from outside the process (like a "normal" database engine)

property Synchronous: TSQLSynchronousMode read GetSynchronous write SetSynchronous;
  *Query or change the SQLite3 file-based synchronization mode, i.e. the way it waits for the data to be flushed on hard drive*
  - default smFull is very slow, but achieve 100% ACID behavior
  - smNormal is faster, and safe until a catastrophic hardware failure occurs
  - smOff is the fastest, data should be safe if the application crashes, but database file may be corrupted in case of failure at the wrong time

TSQLDBSQLite3Statement = class(TSQLDBStatement)
  *Implements a statement using the SQLite3 engine*
  
  *Used for DI-2.2.1 (page 2548).*
constructor Create(aConnection: TSQLDBConnection); override;

Create a SQLite3 statement instance, from an existing SQLite3 connection
- the Execute method can be called once per TSQLDBSQLite3Statement instance, but you can use the Prepare once followed by several ExecutePrepared methods
- if the supplied connection is not of TOleDBConnection type, will raise an exception

destructor Destroy; override;

Release all associated memory and SQLite3 handles

function ColumnBlob(Col: integer): RawByteString; override;

Return a Column as a blob value of the current Row, first Col is 0
- ColumnBlob() will return the binary content of the field is was not ftBlob, e.g. a 8 bytes RawByteString for a vtInt64/vtDouble/vtDate/vtCurrency, or a direct mapping of the RawUnicode

function ColumnCurrency(Col: integer): currency; override;

Return a Column currency value of the current Row, first Col is 0
- should retrieve directly the 64 bit Currency content, to avoid any rounding/conversion error from floating-point types

function ColumnDateTime(Col: integer): TDateTime; override;

Return a Column floating point value of the current Row, first Col is 0

function ColumnDouble(Col: integer): double; override;

Return a Column floating point value of the current Row, first Col is 0

function ColumnIndex(const aColumnName: RawUTF8): integer; override;

Returns the Column index of a given Column name
- Columns numeration (i.e. Col value) starts with 0
- returns -1 if the Column name is not found (via case insensitive search)

function ColumnInt(Col: integer): Int64; override;

Return a Column integer value of the current Row, first Col is 0

function ColumnName(Col: integer): RawUTF8; override;

Retrieve a column name of the current Row
- Columns numeration (i.e. Col value) starts with 0
- it's up to the implementation to ensure than all column names are unique

function ColumnNull(Col: integer): boolean; override;

Returns TRUE if the column contains NULL

function ColumnType(Col: integer; FieldSize: PInteger=nil): TSQLDBFieldType; override;

The Column type of the current Row
- ftCurrency type should be handled specifically, for faster process and avoid any rounding issue, since currency is a standard OleDB type

function ColumnUTF8(Col: integer): RawUTF8; override;

Return a Column UTF-8 encoded text value of the current Row, first Col is 0
**function** Step(SeekFirst: boolean=false): boolean; **override**

After a statement has been prepared via Prepare() + ExecutePrepared() or Execute(), this method must be called one or more times to evaluate it
- you shall call this method before calling any Column*() methods
- return TRUE on success, with data ready to be retrieved by Column*()
- return FALSE if no more row is available (e.g. if the SQL statement is not a SELECT but an UPDATE or INSERT command)
- access the first or next row of data from the SQL Statement result: if SeekFirst is TRUE, will put the cursor on the first row of results, otherwise, it will fetch one row of data, to be called within a loop
- raise an ESQLite3Exception exception on any error

**function** UpdateCount: integer; **override**

Gets a number of updates made by latest executed statement

**procedure** Bind(Param: Integer; Value: Int64; IO: TSQLDBParamInOutType=paramIn); overload; **override**

Bind an integer value to a parameter
- the leftmost SQL parameter has an index of 1

**procedure** Bind(Param: Integer; Value: double; IO: TSQLDBParamInOutType=paramIn); overload; **override**

Bind a double value to a parameter
- the leftmost SQL parameter has an index of 1

**procedure** BindBlob(Param: Integer; Data: pointer; Size: integer; IO: TSQLDBParamInOutType=paramIn); overload; **override**

Bind a Blob buffer to a parameter
- the leftmost SQL parameter has an index of 1

**procedure** BindBlob(Param: Integer; const Data: RawByteString; IO: TSQLDBParamInOutType=paramIn); overload; **override**

Bind a Blob buffer to a parameter
- the leftmost SQL parameter has an index of 1

**procedure** BindCurrency(Param: Integer; Value: currency; IO: TSQLDBParamInOutType=paramIn); overload; **override**

Bind a currency value to a parameter
- the leftmost SQL parameter has an index of 1

**procedure** BindDateTime(Param: Integer; Value: TDateTime; IO: TSQLDBParamInOutType=paramIn); overload; **override**

Bind a TDateTime value to a parameter
- the leftmost SQL parameter has an index of 1

**procedure** BindNull(Param: Integer; IO: TSQLDBParamInOutType=paramIn; BoundType: TSQLDBFieldType=ftNull); **override**

Bind a NULL value to a parameter
- the leftmost SQL parameter has an index of 1

**procedure** BindTextP(Param: Integer; Value: PUTF8Char; IO: TSQLDBParamInOutType=paramIn); overload; **override**

Bind a UTF-8 encoded buffer text (#0 ended) to a parameter
- the leftmost SQL parameter has an index of 1
procedure BindTextS(Param: Integer; const Value: string; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a UTF-8 encoded string to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindTextU(Param: Integer; const Value: RawUTF8; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a UTF-8 encoded string to a parameter
- the leftmost SQL parameter has an index of 1

procedure BindTextW(Param: Integer; const Value: WideString; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a UTF-8 encoded string to a parameter
- the leftmost SQL parameter has an index of 1

procedure ColumnsToJSON(WR: TJSONWriter); override;

Append all columns values of the current Row to a JSON stream
- will use WR.Expand to guess the expected output format
- fast overridden implementation with no temporary variable
- BLOB field value is saved as Base64, in the "\uFFF0base64encodedbinary" format and contains true BLOB data

procedure ExecutePrepared; override;

Execute a prepared SQL statement
- parameters marked as ? should have been already bound with Bind*() functions
- raise an ESQLDBException on any error

procedure Prepare(const aSQL: RawUTF8; ExpectResults: Boolean=false); overload; override;

Prepare an UTF-8 encoded SQL statement
- parameters marked as ? will be bound later, before ExecutePrepared call
- if ExpectResults is TRUE, then Step() and Column*() methods are available to retrieve the data rows
- raise an ESQLDBException on any error

procedure ReleaseRows; override;

Finalize the cursor

procedure Reset; override;

Reset the previous prepared statement

Functions or procedures implemented in the SynDBSQLite3 unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RowsToSQLite3</td>
<td>Direct export of a DB statement rows into a SQLite3 database</td>
<td>1303</td>
</tr>
</tbody>
</table>

function RowsToSQLite3(const Dest: TFileName; const TableName: RawUTF8; Rows: TSQLDBStatement; UseMormotCollations: boolean): integer;

Direct export of a DB statement rows into a SQLite3 database
- the corresponding table will be created within the specified DB file
27.17. SynDBVCL.pas unit

Purpose: DB VCL read/only dataset from SynDB data access
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the SynDBVCL unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynVirtualDataSet</td>
<td>DB VCL read-only virtual dataset</td>
<td>1840</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

SynDBVCL class hierarchy

![Class Hierarchy Diagram]

Objects implemented in the SynDBVCL unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSynBinaryDataSet</td>
<td>Read-only virtual TDataSet able to access a binary buffer as returned by TSQLStatement.FetchAllToBinary method or directly a TSQLStatement</td>
<td>1304</td>
</tr>
<tr>
<td>TSynDBSQLDataSet</td>
<td>TDataSet able to execute any SQL as SynDB’s TSQLStatement result set</td>
<td>1305</td>
</tr>
</tbody>
</table>

TSynBinaryDataSet = class(TSynVirtualDataSet)

Read-only virtual TDataSet able to access a binary buffer as returned by TSQLStatement.FetchAllToBinary method or directly a TSQLStatement

_destructor_ Destroy; _override_; 

Finalize the class instance
procedure From(Statement: TSQLDBStatement; MaxRowCount: cardinal=0; IgnoreColumnDataSize: boolean=false); overload; virtual;

Initialize the virtual TDataSet from a SynDB TSQLDBStatement result set
- the supplied ISQLDBRows instance can safely be freed by the caller, since a private binary copy will be owned by this instance (in Data)
- by default, ColumnDataSize would be computed from the supplied data, unless you set IgnoreColumnDataSize=true to set the value to 0 (and force e.g. SynDBVCL TSynBinaryDataSet.InternalInitFieldDefs define the field as ftDefaultMemo)

procedure From(const BinaryData: RawByteString; DataRowPosition: PCardinalDynArray=nil; IgnoreColumnDataSize: boolean=false); overload; virtual;

Initialize the virtual TDataSet from a FetchAllToBinary() buffer
- by default, ColumnDataSize would be computed from the supplied data, unless you set IgnoreColumnDataSize=true to set the value to 0 (and force e.g. SynDBVCL TSynBinaryDataSet.InternalInitFieldDefs define the field as ftDefaultMemo)

property Data: RawByteString read fData;
Read-only access to the internal binary buffer

property DataAccess: TSQLDBProxyStatementRandomAccess read fDataAccess;
Read-only access to the internal SynDB data

TSynDBSQLDataSet = class(TSynBinaryDataSet)

TSDataSet able to execute any SQL as SynDB’s TSQLStatement result set
- this class is not meant to be used by itself, but via TSynDBDataSet, defined in SynDBMidasVCL.pas, as a data provider able to apply updates to the remote SynDB connection
- typical usage may be for instance over a SynDBRemote connection:
  props := TSQLDWInHTTPConnectionProperties.Create(....);
  ds := TSynDBSQLDataSet.Create(MainForm);
  ds.CommandText := 'select * from people';
  ds.Open;
  // ... use ds
  ds.Close;
  ds.CommandText := 'select * from customer where id=:(10):';
  ds.Open;
  // ... use ds

procedure From(Statement: TSQLDBStatement; MaxRowCount: cardinal=0; IgnoreColumnDataSize: boolean=false); override;

Initialize the internal TDataSet from a SynDB TSQLDBStatement result set
- the supplied TSQLDBStatement can then be freed by the caller, since a private binary copy will be owned by this instance (in fDataSet.Data)
- by default, ColumnDataSize would be computed from the supplied data, unless you set IgnoreColumnDataSize=true to set the value to 0 (and force e.g. SynDBVCL TSynBinaryDataSet.InternalInitFieldDefs define the field as ftDefaultMemo)

property CommandText: string read fCommandText write fCommandText;
The SQL statement to be executed
- since this statement will be executed via Connection.ExecuteInlined, you can specify optionally inlined parameters to this SQL text
property Connection: TSQLDBConnectionProperties read fConnection write fConnection;

The associated connection properties

### Functions or procedures implemented in the SynDBVCL unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryToDataSet</td>
<td>Fetch a SynDB's TSQLDBStatement.FetchAllToBinary buffer into a VCL DataSet</td>
<td>1306</td>
</tr>
<tr>
<td>ToDataSet</td>
<td>Fetch a SynDB's TSQLDBStatement result into a VCL DataSet</td>
<td>1306</td>
</tr>
<tr>
<td>ToDataSet</td>
<td>Fetch a SynDB's TQuery result into a VCL DataSet</td>
<td>1306</td>
</tr>
<tr>
<td>ToDataSet</td>
<td>Fetch a SynDB ISQLDBRows result set into a VCL DataSet</td>
<td>1306</td>
</tr>
</tbody>
</table>

#### function BinaryToDataSet(aOwner: TComponent; const aBinaryData: RawByteString): TSynBinaryDataSet;

- Fetch a SynDB's TSQLDBStatement.FetchAllToBinary buffer into a VCL DataSet
- just a wrapper around TSynBinaryDataSet.Create + Open
- if you need a writable TDataSet, you can use the slower ToClientDataSet() function as defined in SynDBMidasVCL.pas

#### function ToDataSet(aOwner: TComponent; aStatement: ISQLDBRows; aMaxRowCount: integer=0): TSynBinaryDataSet; overload;

- this overloaded function can use directly a result of the TSQLDBConnectionProperties.Execute() method, as such:

  ```delphi
ds1.DataSet := ToDataSet(self, props.Execute('select * from table',[]));
```

#### function ToDataSet(aOwner: TComponent; aStatement: SynDB.TQuery; aMaxRowCount: integer=0): TSynBinaryDataSet; overload;

- Fetch a SynDB's TQuery result into a VCL DataSet
- if aMaxRowCount>0, will return up to the specified number of rows
- current implementation will return a TSynSQLStatementDataSet instance, using an optimized internal binary buffer: the supplied TQuery can be released
- if you need a writable TDataSet, you can use the slower ToClientDataSet() function as defined in SynDBMidasVCL.pas

#### function ToDataSet(aOwner: TComponent; aStatement: TSQLDBStatement; aMaxRowCount: integer=0): TSynBinaryDataSet; overload;

- Fetch a SynDB's TSQLDBStatement result into a VCL DataSet
- just a wrapper around TSynSQLStatementDataSet.Create + Open
- if aMaxRowCount>0, will return up to the specified number of rows
- current implementation will return a TSynSQLStatementDataSet instance, using an optimized internal binary buffer: the supplied statement can be released
- if you need a writable TDataSet, you can use the slower ToClientDataSet() function as defined in SynDBMidasVCL.pas
27.18. SynDBZeos.pas unit

*Purpose*: ZEOS 7.x direct access classes for SynDB units (not DB.pas based)
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the *SynDBZeos* unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache-buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
</tbody>
</table>

*SynDBZeos class hierarchy*

Objects implemented in the *SynDBZeos* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESQLDBZEOS</td>
<td>Exception type associated to the ZEOS database components</td>
<td>1308</td>
</tr>
<tr>
<td>TSQLDBZEOSConnection</td>
<td>Implements a connection via the ZEOS access layer</td>
<td>1310</td>
</tr>
<tr>
<td>TSQLDBZEOSConnectionProperties</td>
<td>Implement properties shared by ZEOS connections</td>
<td>1308</td>
</tr>
<tr>
<td>TSQLDBZEOSStatement</td>
<td>Implements a statement via a ZEOS database connection</td>
<td>1310</td>
</tr>
</tbody>
</table>
ESQLDBZEOS = class(ESQLDBException)
    Exception type associated to the ZEOS database components

TSQLDBZEOSConnectionProperties = class(TSQLDBConnectionPropertiesThreadSafe)
    Implement properties shared by ZEOS connections

constructor Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8);
    Initialize the properties to connect to the ZEOS engine
    - aServerName shall contain the ZEOS URI, e.g:
        zdbc:firebird-2.0://127.0.0.1:3050/model?username=sysdba;password=masterkey
        zdbc:mysql://192.168.2.60:3306/world?username=root;password=dev
        sqlite
    i.e. '[zdbc:]PROTOCOL://HOST:PORT[/DATABASE][?paramname=value]' 
    - you can define the TZConnection.LibraryLocation property by setting a '?LibLocation=...' parameter within the aServerName URL value 
    - or simply use TSQLDBZEOSConnectionProperties.URI() class method
    - aDatabaseName, aUserID, aPassword are used if not already set as URI in aServerName value
    - you can use Protocols property to retrieve all available protocol names 
    - note that when run from mORMot's ORM, this class will by default create one connection per thread, which makes some clients (e.g. PostgreSQL) unstable and consuming a lot of resources - you should better maintain one single connection, by setting after Create:
        aExternalDBProperties.ThreadingMode := tmMainConnection;
    or by adding 'syndb_singleconnection=true' as URI property

constructor CreateWithZURL(const aURL: TZURL; aDBMS: TSQLDBDefinition; aOwnsURL: Boolean);
    Initialize raw properties to connect to the ZEOS engine 
    - using Zeos' TZURL detailed class - see: src\core\ZURL.pas 
    - this gives all possibilities to add Properties before a connection is opened 
    - you can define the protocol by hand eg. "odbc_w"/"OleDb" and define TSQLDBDefinition to describe the server syntax SynDB and the ORM use behind the abstract driver

destructor Destroy; override;
    Finalize properties internal structures

function GetDatabaseMetadata(out meta: IZDatabaseMetadata): boolean;
    Access to the database metadata, as retrieved by ZEOS 
    - returns TRUE if metadata interface has been retrieved

function NewConnection: TSQLDBConnection; override;
    Create a new connection 
    - caller is responsible of freeing this instance
    - this overridden method will create an TSQLDBZEOSConnection instance
class function URI(aServer: TSQLDBDefinition; const aServerName: RawUTF8; const aLibraryLocation: TFileName=''; aLibraryLocationAppendExePath: boolean=true): RawUTF8; overload;

  Compute the ZEOS URI for a given database engine
  - the optional server name can contain a port number, specified after ':'
  - you can set an optional full path to the client library name, to be completed on the left side with the executable path
  - possible use may be:
    PropsOracle := TSQLDBZEOSConnectionProperties.Create(
      TSQLDBZEOSConnectionProperties.URI(dOracle,'', 'oci64\oci.dll'),
      'tnsname','user','pass');
    PropsFirebird := TSQLDBZEOSConnectionProperties.Create(
      TSQLDBZEOSConnectionProperties.URI(dFirebird,'', 'Firebird\fbembed.dll'),
      'databasefilename','','');
    PropsFirebird := TSQLDBZEOSConnectionProperties.Create(
      TSQLDBZEOSConnectionProperties.URI(dFirebird,'192.168.1.10:3055',
        'c:\firebird_2_5\bin\fbclient.dll',false),
      '3camadas','sysdba','masterkey');

class function URI(const aProtocol, aServerName: RawUTF8; const aLibraryLocation: TFileName=''; aLibraryLocationAppendExePath: boolean=true): RawUTF8; overload;

  Compute the ZEOS URI for a given protocol
  - if a TSQLDBDefinition may have several protocols (e.g. MSSQL), you can use this overloaded method to select the exact protocol to use if the default one fixed by TSQLDBDefinition does not match your needs
  - the protocol name should contain the trailing : character, e.g. 'firebird-2.0:' if the default 'firebird-2.5:' is not correct

procedure GetFields(const aTableName: RawUTF8; out Fields: TSQLDBColumnDefineDynArray); override;

  Retrieve the column/field layout of a specified table
  - this overridden method will use ZDBC metadata to retrieve the information

procedure GetTableNames(out Tables: TRawUTF8DynArray); override;

  Get all table names
  - this overridden method will use ZDBC metadata to retrieve the information
  - PostgreSQL note: it was reported that some table names expects to be quoted for this DB engine - and ZDBC won't do it for yourself - please ensure you specify the correct quoted table name e.g. when you register the external PostgreSQL table via function VirtualTableExternalRegister()

property DBMSName: RawUTF8 read fDBMSName;

  The remote DBMS name, as retrieved from ServerName, i.e. ZEOS URL

property SupportsArrayBindings: boolean read fSupportsArrayBindings;

  If the associated ZDBC provider supports parameters array binding
  - you should use the BindArray() methods only if this property is TRUE

property ZeosStatementParams: TStrings read fStatementParams;

  Direct access to the internal statement parameters
  - i.e. will be used by IZConnection.PrepareStatementWithParams()
  - default values (set in Create method) try to achieve best performance
property ZeosURL: TZURL read fURL;
    Direct access to the internal TZURL connection parameters

TSQLDBZEOSConnection = class(TSQLDBConnectionThreadSafe)
    Implements a connection via the ZEOS access layer

constructor Create(aProperties: TSQLDBConnectionProperties); override;
    Prepare a connection to a specified ZEOS database server

function IsConnected: boolean; override;
    Return TRUE if Connect has been already successfully called

function NewStatement: TSQLDBStatement; override;
    Create a new statement instance

procedure Commit; override;
    Commit changes of a Transaction for this connection
    - StartTransaction method must have been called before

procedure Connect; override;
    Connect to the specified ZEOS server
    - should raise an ESQLDBZEOS on error

procedure Disconnect; override;
    Stop connection to the specified ZEOS database server
    - should raise an ESQLDBZEOS on error

procedure Rollback; override;
    Discard changes of a Transaction for this connection
    - StartTransaction method must have been called before

procedure StartTransaction; override;
    Begin a Transaction for this connection

property Database: IZConnection read fDatabase;
    Access to the associated ZEOS connection instance

TSQLDBZEOSStatement = class(TSQLDBStatementWithParamsAndColumns)
    Implements a statement via a ZEOS database connection

function ColumnBlob(Col: Integer): RawByteString; override;
    Return a Column as a blob value of the current Row, first Col is 0

function ColumnCurrency(Col: Integer): currency; override;
    Return a Column currency value of the current Row, first Col is 0

function ColumnDateTime(Col: Integer): TDateTime; override;
    Return a Column date and time value of the current Row, first Col is 0

function ColumnDouble(Col: Integer): double; override;
    Return a Column floating point value of the current Row, first Col is 0
function ColumnInt(Col: Integer): Int64; override;
  Return a Column integer value of the current Row, first Col is 0

function ColumnNull(Col: Integer): boolean; override;
  Returns TRUE if the column contains NULL

function ColumnUTF8(Col: Integer): RawUTF8; override;
  Return a Column UTF-8 encoded text value of the current Row, first Col is 0

function Step(SeekFirst: boolean = false): boolean; override;
  Access the next or first row of data from the SQL Statement result
  - return true on success, with data ready to be retrieved by Column*() methods
  - return false if no more row is available (e.g. if the SQL statement is not a SELECT but an UPDATE or INSERT command)
  - if SeekFirst is TRUE, will put the cursor on the first row of results
  - raise an ESQLDBZeos on any error

function UpdateCount: integer; override;
  Gets a number of updates made by latest executed statement

procedure ColumnsToJSON(WR: TJSONWriter); override;
  Append all columns values of the current Row to a JSON stream
  - will use WR.Expand to guess the expected output format
  - this overridden implementation will call fReultSet methods to avoid creating most temporary variable

procedure ExecutePrepared; override;
  Execute a prepared SQL statement
  - parameters marked as ? should have been already bound with Bind*() functions
  - this implementation will also handle bound array of values (if any), if IZDatabaseInfo.SupportsArrayBindings is true for this provider
  - this overridden method will log the SQL statement if sllSQL has been enabled in SynDBLog.Family.Level
  - raise an ESQLDBZeos on any error

procedure Prepare(const aSQL: RawUTF8; ExpectResults: boolean = false); overload; override;
  Prepare an UTF-8 encoded SQL statement
  - parameters marked as ? will be bound later, before ExecutePrepared call
  - if ExpectResults is TRUE, then Step() and Column*() methods are available to retrieve the data rows
  - raise an ESQLDBZeos on any error

procedure ReleaseRows; override;
  Free IZResultSet/IZResultSetMetaData when ISQLDBStatement is back in cache

procedure Reset; override;
  Reset the previous prepared statement
  - this overridden implementation will reset all bindings and the cursor state
  - raise an ESQLDBZeos on any error
property JSONComposeOptions: TZJSONComposeOptions read fJSONComposeOptions write fJSONComposeOptions default [jcoEndJSONObject];

    The ColumnsToJSON options provided by ZDBC
    - jcoEndJSONObject: cancels last comma, adds the close object bracket '}' and add's the next
        comma. If not set you can continue writting some custom data into your object json but you also
        have to finalize each row-object then.
    - jcoMongoISOString: formats the date,time,datetime values as mongo
        ISODate("YYYY-MM-DDTHH:NN:ssZ"). Milliseconds are included. So the values are recognized as
        date type by mongodb. Otherwise mongo threads them as strings. This option might be usefull if
        you export sql rows using a json streamed file which will be used as an import-file with your
        mongodb. If set the options jcoDATETIME_MAGIC and jcoMilliseconds are ignored
    - jcoDATETIME_MAGIC add the JSON_SQLDATE_MAGIC on top of data before adding the ISO
        date,time,datetime value quoted strings
    - jcoMilliseconds compose the time/datetime values with milliseconds
    - jcsSkipNulls ignore null columns. So neither fieldname nor the null value will be composed into
        your JSON. For real big JSON contents it saves loads of space. e.g. if you import a JSON into a
        mongo cluster you'll have a significant space difference if null's are simply ignored.

Functions or procedures implemented in the SynDBZeos unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetZEOSProtocols</td>
<td>To be called in order to populate the global ZEOSProtocols list</td>
<td>1312</td>
</tr>
</tbody>
</table>

procedure SetZEOSProtocols;

    To be called in order to populate the global ZEOSProtocols list

Variables implemented in the SynDBZeos unit

ZEOSProtocols: TRawUTF8DynArray;

    List of all available ZEOS protocols
    - you have to call SetZEOSProtocols before using it, to update this global list with all initialized
        ZPlain*Driver units
    - to be used e.g. within ZEOS URI, as TSQLDBZEOSConnectionProperties.ServerName
27.19. SynEcc.pas unit

Purpose: Certificate-based public-key cryptography using ECC-secp256r1
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the SynEcc unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
<tr>
<td></td>
<td>- implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

Objects implemented in the SynEcc unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EECCException</td>
<td>Exception class associated with this SynEcc unit</td>
<td>1318</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TECCCertificate</td>
<td>A public certificate using ECC secp256r1 cryptography</td>
<td>1318</td>
</tr>
<tr>
<td>TECCCertificateChain</td>
<td>Manage PKI certificates using ECC secp256r1 cryptography</td>
<td>1329</td>
</tr>
<tr>
<td>TECCCertificateContent</td>
<td>Store a TECCCertificate binary buffer for ECC secp256r1 cryptography</td>
<td>1315</td>
</tr>
<tr>
<td>TECCCertificateDecryptSetting</td>
<td>Store settings pointing to a local .private file containing a secret key for .synecc file decryption</td>
<td>1326</td>
</tr>
<tr>
<td>TECCCertificateSecret</td>
<td>A public/private certificate using ECC secp256r1 cryptography</td>
<td>1321</td>
</tr>
<tr>
<td>TECCCertificateSecretSetting</td>
<td>Store settings pointing to a local .private file containing a secret key</td>
<td>1325</td>
</tr>
<tr>
<td>TECCCertificateSigned</td>
<td>The certification information of a TECCCertificate</td>
<td>1314</td>
</tr>
<tr>
<td>TECCSignatureCertified</td>
<td>A ECDSA secp256r1 digital signature of some content, signed by an authority</td>
<td>1326</td>
</tr>
<tr>
<td>TECCSignatureCertifiedContent</td>
<td>Store a TECCSignatureCertified binary buffer for ECDSA secp256r1 signature</td>
<td>1315</td>
</tr>
<tr>
<td>TECCSignatureCertifiedFile</td>
<td>Handle a .sign file content as generated by TECCCertificateSecret.SignFile</td>
<td>1328</td>
</tr>
<tr>
<td>TECDEAlgo</td>
<td>Defines one protocol Algorithm recognized by TECDEProtocol</td>
<td>1317</td>
</tr>
<tr>
<td>TECDEFrameClient</td>
<td>The binary handshake message, sent by client to server</td>
<td>1317</td>
</tr>
<tr>
<td>TECDEFrameServer</td>
<td>The binary handshake message, sent back from server to client</td>
<td>1318</td>
</tr>
<tr>
<td>TECDEProtocol</td>
<td>Abstract ECDHE secure protocol with unilateral or mutual authentication</td>
<td>1335</td>
</tr>
<tr>
<td>TECDEProtocolClient</td>
<td>Implements ECDHE secure protocol on client side</td>
<td>1337</td>
</tr>
<tr>
<td>TECDEProtocolServer</td>
<td>Implements ECDHE secure protocol on server side</td>
<td>1337</td>
</tr>
<tr>
<td>TECIESHeader</td>
<td>Binary header of a .synecc file, encrypted via ECC secp256r1</td>
<td>1316</td>
</tr>
<tr>
<td>TJWTES256</td>
<td>Implements JSON Web Tokens using 'ES256' algorithm</td>
<td>1338</td>
</tr>
</tbody>
</table>

**TECCCertificateSigned = packed record**

*The certification information of a TECCCertificate*
- as stored in TECCCertificateContent.Signed
- defined in a separate record, to be digitally signed in the Signature field
- map TECCCertificate.Version 1 of the binary format
- "self-signed" certificates may be used as "root" certificates in the TECCCertificateChain list

**AuthorityIssuer: TECCCertificateIssuer;**

*Identify the authority issuer used for signing*
- is either genuine random bytes, or some Baudot-encoded text
- may equal Issuer, if was self-signed
AuthoritySerial: TECCCertificateID;
   Genuine identifier of the authority certificate used for signing
   - should be used to retrieve the associated PublicKey used to compute the Signature field
   - may equal Serial, if was self-signed

IssueDate: TECCDate;
   When this certificate was generated

Issuer: TECCCertificateIssuer;
   Identify the certificate issuer
   - is either genuine random bytes, or some Baudot-encoded text

PublicKey: TECCPublicKey;
   The ECDSA secp256r1 public key of this certificate
   - may be used later on for signing or key derivation

Serial: TECCCertificateID;
   A genuine identifier for this certificate
   - is used later on to validate other certificates in chain

ValidityEnd: TECCDate;
   Certificate valid not after

ValidityStart: TECCDate;
   Certificate valid not before

TECCCertificateContent = packed record
   Store a TECCCertificate binary buffer for ECC secp256r1 cryptography
   - i.e. a certificate public key, with its ECDSA signature
   - would be stored in 173 bytes

CRC: cardinal;
   FNV-1a checksum of all previous fields
   - we use fnv32 and not crc32c here to avoid colision with crc64c hashing
   - avoiding to compute slow ECDSA verification in case of corruption, due e.g. to unexpected
     transmission/bug/fuzzing
   - should be the very last field in the record

Signature: TECCSignature;
   SHA-256 + ECDSA secp256r1 signature of the Certificate record

Signed: TECCCertificateSigned;
   The certification information, digitally signed in the Signature field

Version: word;
   The TECCCertificate format version

TECCSignatureCertifiedContent = packed record
   Store a TECCSignatureCertified binary buffer for ECDSA secp256r1 signature
   - i.e. the digital signature of some content
AuthorityIssuer: TECCCertificateIssuer;
  
  Identify the authority issuer used for signing
  - is either genuine random bytes, or some Baudot-encoded text

AuthoritySerial: TECCCertificateID;

  Genuine identifier of the authority certificate used for signing
  - should be used to retrieve the associated PublicKey used to compute the Signature field

Date: TECCDate;
  
  When this signature was generated

Signature: TECCSignature;
  
  SHA-256 + ECDSA secp256r1 digital signature of the content

Version: word;
  
  The TECCSignatureCertificated format version

TECIESHeader = packed record
  
  Binary header of a .synecc file, encrypted via ECC secp256r1
  - as generated by TECCCertificate.Encrypt/EncryptFile, and decoded by
  TECCCertificateSecret.Decrypt
  - a sign-then-encrypt pattern may have been implemented for additional safety

algo: TECIESAlgo;
  
  Actual encryption algorithm used

crc: cardinal;
  
  A crc32c hash of the header (excluding this field)

date: TECCDate;
  
  When this encryption was performed

hmac: THash256;
  
  The Message Authentication Code of the encrypted content

magic: THash128;
  
  Contains 'SynEccEncrypted'#26
  - so every .synecc file starts with those characters as signature

rec: TECCCertificateIssuer;
  
  TECCCertificate.Issuer of the recipient public key used for encryption
  - is either genuine random bytes, or some Baudot-encoded text

recid: TECCCertificateID;
  
  TECCCertificate.Serial of the recipient public key used for encryption

rndpub: TECCPublicKey;
  
  The genuine random public key used for encryption

sign: TECCSignatureCertifiedContent;
  
  Optional ECDSA secp256r1 digital signature of the plain content

size: cardinal;
  
  The size of the plain content (may be compressed before encryption)
unixts: cardinal;
Optional timestamp, in Unix seconds since 1970, of the source file

**TECDHEAlgo** = packed record

Defines one protocol Algorithm recognized by TECDEHProtocol
- only safe and strong parameters are allowed, and the default values (i.e. all fields set to 0) will ensure a very good combination
- in current implementation, there is no negotiation between nodes: client and server should have the very same algorithm

auth: TECDEHAuth;
The current Authentication scheme
ef: TECDEHEF;
The current Encryption Function
kdf: TECDEHKDF;
The current Key Derivation Function
mac: TECDEMAC;
The current Message Authentication Code

**TECDHEFrameClient** = packed record

The binary handshake message, sent by client to server
- the frame will always have the same fixed size of 290 bytes (i.e. 388 base64-encoded chars, which could be transmitted in a HTTP header), for both mutual or unilateral authentication
- ephemeral keys may be included for perfect forward security

algo: TECDEHAlgo;
Expected algorithm used
QCA: TECCCertificateContent;
Client public key, with its certificate
- may be zero, in case of unilateral authentication (algo=authServer)
QE: TECCPublicKey;
Client-generated ephemeral public key
- may be zero, in case of unilateral authentication (algo=authClient)
RndA: THash128;
A client-generated random seed
Sign: TECCSignature;
SHA-256 + ECDSA secp256r1 signature of the previous fields, computed with the client private key
- i.e. ECDSASign(dA,sha256(algo|RndA|QCA|QE))
- may be zero, in case of unilateral authentication (algo=authServer)
TECDHEFrameServer = packed record

The binary handshake message, sent back from server to client
- the frame will always have the same fixed size of 306 bytes (i.e. 408 base64-encoded chars, which could be transmitted in a HTTP header), for both mutual or unilateral authentication
- ephemeral keys may be included for perfect forward security

algo: TECDHEAlgo;
  Algorithm used by the server

QCB: TECCCertificateContent;
  Server public key, with its certificate
  - may be zero, in case of unilateral authentication (algo=authClient)

QF: TECCPublicKey;
  Server-generated ephemeral public key
  - may be zero, in case of unilateral authentication (algo=authServer)

RndA: THash128;
  Client-generated random seed

RndB: THash128;
  A server-generated random seed

Sign: TECCSignature;
  SHA-256 + ECDSA secp256r1 signature of the previous fields, computed with the server private key
  - i.e. ECDSASign(dB,sha256(algo|RndA|RndB|QCB|QF))
  - may be zero, in case of unilateral authentication (algo=authClient)

EECCException = class(ESynException)
  Exception class associated with this SynEcc unit

TECCCertificate = class(TSynPersistent)

A public certificate using ECC secp256r1 cryptography
- implements a custom binary format, with validation period, and chaining
- could be used for safe data signing, and authentication
- in fact, Base64 published property is enough to persist this instance: but consider also
  ToBase64/FromBase64/LoadFromStream/SaveToStream methods

constructor Create; override;
  Initialize this certificate

constructor CreateFrom(const binary: TECCCertificateContent); virtual;
  Initialize this certificate from a supplied certificate binary
  - will raise an EECCException if the supplied binary is incorrect
constructor CreateFromAuth(const AuthPubKey: TFileName; const AuthBase64, AuthSerial: RawUTF8); virtual;

*Initialize this certificate from a set of potential inputs*
- will first search from a .public file name, base-64 encoded binary, or a serial number which be used to search for a local .public file (as located by ECCKeyFileFind)
- will raise an EECCException if no supplied media is correct

constructor CreateFromBase64(const base64: RawUTF8); virtual;

*Initialize this certificate from a supplied base-64 encoded binary*
- will raise an EECCException if the supplied base64 is incorrect

function CheckCRC: boolean;

*Fast check of the binary buffer storage of this certificate*
- ensure Content.CRC has the expected value, using FNV-1a checksum
- does not validate the certificate against the certificates chain, nor perform any ECC signature: use TECCCertificateChain.IsValid instead

function Encrypt(const Plain: RawByteString; Signature: TECCSSignatureCertified=nil; FileDateTime: TDateTime=0; const KDFSalt: RawUTF8='salt'; KDFRounds: integer=DEFAULT_ECCROUNDS; const MACSalt: RawUTF8='hmac'; MACRounds: integer=100; Algo: TECIESAlgo=ecaUnknown): RawByteString;

*Encrypt using the ECIES scheme, using this public certificate as key, via AES-256-CFB/PKCS7 over PBKDF2_HMAC_SHA256, and HMAC_SHA256*
- returns the encrypted content, in the .synecc optimized format
- optional salt information used for PBKDF2 or HMAC can be customized
- ecaUnknown algorithm will use either ecaPBKDF2_HMAC_SHA256_AES256_CFB or ecaPBKDF2_HMAC_SHA256_AES256_CFB_SYNLZ depending if the supplied contain is compressible or not - but you may force another algorithm
- you can optionally associate an ECDSA secp256r1 digital signature, and a timestamp which may be used when re-creating a decyphered file
- use TECCCertificateSecret.Decrypt to uncrypher the resulting content

function EncryptFile(const FileToCrypt: TFileName; const DestFile: TFileName=''; const Salt: RawUTF8='salt'; SaltRounds: integer=DEFAULT_ECCROUNDS; Algo: TECIESAlgo=ecaUnknown; IncludeSignFile: boolean=true): boolean;

*Encrypt a file using the ECIES scheme, using this public certificate as key, via AES-256-CFB/PKCS7 over PBKDF2_HMAC_SHA256, and HMAC_SHA256*
- by default, will create a FileToCrypt.synecc encrypted file
- ecaUnknown algorithm will use either ecaPBKDF2_HMAC_SHA256_AES256_CFB or ecaPBKDF2_HMAC_SHA256_AES256_CFB_SYNLZ depending if the supplied contain is compressible or not - but you may force another algorithm
- any available .sign ECDSA secp256r1 digital signature file will be recognized and embedded to the resulting .synecc content
- optional salt information used for PBKDF2 can be customized, to lock the encryted file with the supplied password
**function** FromAuth(const AuthPubKey: TFileName; const AuthBase64, AuthSerial: RawUTF8): boolean;

*Retrieve the certificate from a set of potential inputs*
- will first search from a .public file name, base-64 encoded binary, or a serial number which be used to search for a local .public file in the current folder or ECCKeyFileFolder (as located by ECCKeyFileFind)
- returns true on success, false otherwise

**function** FromBase64(const base64: RawUTF8): boolean;

*Retrieve the certificate from some base-64 encoded binary*
- will use LoadFromStream serialization
- returns true on success, false otherwise

**function** FromFile(const filename: TFileName): boolean;

*Retrieve the certificate from the "Base64": JSON entry of a .public file*
- will use FromBase64/LoadFromStream serialization
- returns true on success, false otherwise

**function** LoadFromStream(Stream: TStream): boolean;

*Persist only the public certificate as some base-64 encoded binary*
- will follow TECCertificate.SaveToStream/ToBase64 serialization, even when called from a TECCCertificateSecret instance
- could be used to safely publish the public information of a newly created certificate

**function** SaveToStream(Stream: TStream): boolean;

*Persist the certificate as some binary*
- returns true on success (i.e. this class stores a certificate), false otherwise

**function** ToBase64: RawUTF8;

*Save the public key as a .public json file*
- i.e. a json containing all published properties of this instance
- persist ToVariant() as an human-readable JSON file

**function** ToFile(const filename: TFileName): boolean;

*Saves the public key as a .public file*
- i.e. a json containing all published properties of this instance
- excludes the Base64 property content if withBase64 is set to false

**property** AuthorityIssuer: RawUTF8 read GetAuthorityIssuer;

*Identify the authority issuer used for signing, as text*

**property** AuthoritySerial: RawUTF8 read GetAuthoritySerial;

*Hexadecimal text of the authority certificate identifier used for signing*
property Base64: RawUTF8 read PublicToBase64 write SetBase64;
  Base-64 encoded text of the whole certificate binary information
  - only the public part of the certificate will be shown: any private key of a TECCCertificateSecret instance would be trimmed

property Content: TECCCertificateContent read fContent write fContent;
  Low-level access to the binary buffer used ECC secp256r1 cryptography
  - you should not use this property, but other methods

property IsSelfSigned: boolean read GetIsSelfSigned;
  If this certificate has been signed by itself
  - a self-signed certificate will have its AuthoritySerial/AuxiliaryIssuer fields matching Serial/Issuer, and should be used as "root" certificates

property IssueDate: RawUTF8 read GetIssueDate;
  When this certificate was generated, as ISO-8601 text

property Issuer: RawUTF8 read GetIssuer;
  Identify the certificate issuer, as text

property Serial: RawUTF8 read GetSerial;
  The genuine identifier of this certificate, as hexadecimal text

property Signature: TECCCertificateContent read fContent.Signature;
  SHA-256 + ECDSA secp256r1 signature of the Certificate record

property Signed: TECCCertificateContent read fContent.Signed;
  The certification information, digitally signed in the Signature field

property ValidityEnd: RawUTF8 read GetValidityEnd;
  Valid not after this date, as ISO-8601 text

property ValidityStart: RawUTF8 read GetValidityStart;
  Valid not before this date, as ISO-8601 text

property Version: word read fContent.Version;
  The TECCCertificate format version
  - currently equals 1

TECCCertificateSecret = class(TECCCertificate)
A public/private certificate using ECC secp256r1 cryptography
  - will store TECCCertificate public and associated private secret key
  - implements a custom binary format, with validation period, and chaining
  - could be used for safe data signing via SignToBase64/SignFile, and authentication / key derivation
  - allows optional anti-forensic diffusion during storage via AFSplitStripes
constructor CreateFromSecureBinary(Data: pointer; Len: integer; const PassWord: RawUTF8; PBKDF2Rounds: integer=DEFAULT_ECCROUNDS; AES: TAESAbstractClass=nil);

  Create a certificate with its private secret key from a password-protected secure binary buffer
  - may be used on a constant array in executable, created via SaveToSource()
  - perform all reverse steps from SaveToSecureBinary() method
  - will raise an EECEException if the supplied Binary is incorrect

constructor CreateFromSecureBinary(const Binary: RawByteString; const PassWord: RawUTF8; PBKDF2Rounds: integer=DEFAULT_ECCROUNDS; AES: TAESAbstractClass=nil);
  Create a certificate with its private secret key from a password-protected secure binary buffer
  - perform all reverse steps from SaveToSecureBinary() method
  - will raise an EECEException if the supplied Binary is incorrect

constructor CreateFromSecureFile(const FolderName: TFileName; const Serial, PassWord: RawUTF8; PBKDF2Rounds: integer=DEFAULT_ECCROUNDS; AES: TAESAbstractClass=nil); overload;
  Create a certificate with its private secret key from an encrypted secure .private binary file stored in a given folder
  - overloaded constructor retrieving the file directly from its folder
  - perform all reverse steps from SaveToSecureFile() method
  - will raise an EECEException if the supplied file is incorrect

constructor CreateFromSecureFile(const FileName: TFileName; const PassWord: RawUTF8; PBKDF2Rounds: integer=DEFAULT_ECCROUNDS; AES: TAESAbstractClass=nil);
  Create a certificate with its private secret key from an encrypted secure .private binary file and its associated password
  - perform all reverse steps from SaveToSecureFile() method
  - will raise an EECEException if the supplied file is incorrect

constructor CreateNew(Authority: TECCCertificateSecret; const IssuerText: RawUTF8=''; ExpirationDays: integer=0; StartDate: TDateTime=0; ParanoidVerify: boolean=true);
  Generate a new certificate, signed using the supplied Authority
  - if Authority is nil, will generate a self-signed certificate
  - the supplied Issuer name would be stored using AsciiToBaudot(), truncated to the Issuer buffer size, i.e. 16 bytes - if Issuer is "", TAESPRNG.Fill() will be used
  - you may specify some validity time range, if needed
  - default ParanoidVerify=true will validate the certificate digital signature via a call ecdsa_verify() to ensure its usefulness
  - would take around 4 ms under a 32-bit compiler, and 1 ms under 64-bit

destructor Destroy; override;
  Finalize the instance, and safe erase fPrivateKey stored buffer
function Decrypt(const Encrypted: RawByteString; out Decrypted: RawByteString;
Signature: PECCSignatureCertifiedContent=nil; MetaData: PRawJSON=nil;
FileDateTime: PDateTime=nil; const KDFSalt: RawUTF8='salt'; KDFRounds:
integer=DEFAULT_ECCROUNDS; const MACSalt: RawUTF8='hmac'; MACRounds: integer=100):
TECCDecrypt;

Decrypt using the ECIES scheme, using this private certificate as key, via AES-256-CFB/PKCS7 over
PBKDF2_HMAC_SHA256, and HMAC_SHA256
- expects TECCCertificate.Crypt() cyphered content with its public key
- returns the decrypted content, or " in case of failure
- optional shared information used for PBKDF2 or HMAC can be customized
- optionally, you can retrieve the sign-then-encrypt ECDSA secp256r1 signature and metadata
stored in the header (to be checked via TECCCertificateChain.IsSigned method), and/or the
associated file timestamp

function DecryptFile(const FileNameToDecrypt: TFileName; const DestFileName='';
const Salt: RawUTF8='salt'; SaltRounds: integer=DEFAULT_ECCROUNDS; Signature:
PECCSignatureCertifiedContent=nil; MetaData: PRawJSON=nil): TECCDecrypt;

Decrypt using the ECIES scheme, using this private certificate as key, decrypt a file using the
ECIES scheme, using this private certificate as key, via AES-256-CFB/PKCS7 over
PBKDF2_HMAC_SHA256, and HMAC_SHA256
- makes the reverse operation of TECCCertificate.EncryptFile method
- by default, will erase the (.synecc) extension to FileNameToDecrypt name
- optional salt information used for PBKDF2 can be customized, to unlock the encrypted file with
the supplied password
- optionally, you can retrieve the sign-then-encrypt ECDSA secp256r1 signature stored in the
header for TECCCertificateChain.IsSigned() in supplied Signature^ and MetaData^ values

function HasSecret: boolean;
Returns TRUE if the private secret key is not filled with zeros

function LoadFromSecureBinary(Data: pointer; Len: integer; const PassWord: RawUTF8;
PBKDF2Rounds: integer=DEFAULT_ECCROUNDS; AES: TAESAbstractClass=nil): boolean;
overload;
Read a private secret key from an encrypted secure binary buffer
- perform all reverse steps from SaveToSecureBinary() method
- returns TRUE on success, FALSE otherwise

function LoadFromSecureBinary(const Binary: RawByteString; const PassWord: RawUTF8;
PBKDF2Rounds: integer=DEFAULT_ECCROUNDS; AES: TAESAbstractClass=nil): boolean;
overload;
Read a private secret key from an encrypted secure binary buffer
- perform all reverse steps from SaveToSecureBinary() method
- returns TRUE on success, FALSE otherwise

function LoadFromSecureFile(const FileName: TFileName; const PassWord: RawUTF8;
PBKDF2Rounds: integer=DEFAULT_ECCROUNDS; AES: TAESAbstractClass=nil): boolean;
Read a private secret key from an encrypted .private binary file
- perform all reverse steps from SaveToSecureBinary() method
- returns TRUE on success, FALSE otherwise
function SaveToSecureBinary(const PassWord: RawUTF8; AFStripes: integer=64; PBKDF2Rounds: integer=DEFAULT_ECCROUNDS; AES: TAESAbstractClass=nil; NoHeader: boolean=false): RawByteString;

Backup the private secret key into an encrypted secure binary buffer
- you should keep all your private keys in a safe place
- will use anti-forensic diffusion of the private key (64 stripes = 2KB)
- then AES-256-CFB encryption (or the one specified in AES parameter) will be performed from PBKDF2_HMAC_SHA256 derivation of an user-supplied password

function SaveToSecureFile(const PassWord: RawUTF8; const DestFolder: TFileName; AFStripes: integer=64; PBKDF2Rounds: integer=DEFAULT_ECCROUNDS; AES: TAESAbstractClass=nil; NoHeader: boolean=false): boolean;

Backup the private secret key into an encrypted .private binary file
- you should keep all your private keys in a safe dedicated folder
- filename will be the certificate hexadecimal as 'Serial.private'
- will use anti-forensic diffusion of the private key (64 stripes = 2KB)
- then AES-256-CFB encryption (or the one specified in AES parameter) will be performed from PBKDF2_HMAC_SHA256 derivation of an user-supplied password

function SaveToSecureFileName(FileNumber: integer=0): TFileName;

Computes the 'Serial.private' file name of this certificate
- as used by SaveToSecureFile()

function SaveToSecureFiles(const PassWord: RawUTF8; const DestFolder: TFileName; DestFileCount: integer; AFStripes: integer=64; PBKDF2Rounds: integer=DEFAULT_ECCROUNDS; AES: TAESAbstractClass=nil; NoHeader: boolean=false): boolean;

Backup the private secret key into several encrypted -###.private binary files
- secret sharing can be used to store keys at many different places, e.g. on several local or remote drives, and therefore enhance privacy and safety
- it will use anti-forensic diffusion of the private key to distribute it into pieces, in a manner that a subset of files can not regenerate the key: as a result, a compromission of one sub-file won't affect the secret key
- filename will be the certificate hexadecimal as 'Serial-###.private'
- AES-256-CFB encryption (or the one specified in AES parameter) will be performed from PBKDF2_HMAC_SHA256 derivation of an user-supplied password

function SaveToSource(const ConstName, Comment, PassWord: RawUTF8; IncludePassword: boolean=true; AFStripes: integer=0; PBKDF2Rounds: integer=100; AES: TAESAbstractClass=nil; IncludeRaw: boolean=true): RawUTF8;

Backup the private secret key into an encrypted source code constant
- may be used to integrate some private keys within an executable
- if ConstName="", _HEXASERIAL will be used, from 24 first chars of Serial
- the password may also be included as ConstName_PASS associated constant, and as ConstName_CYPH in TSynPersistentWithPassword/TECCCertificateSecretSetting encrypted format
function SignFile(const FileToSign: TFileName; const MetaNameValuePairs: array of const): TFileName;

Compute a .sign digital signature of any file
- SHA-256/ECDSA digital signature is included in a JSON document
- you can set some additional metadata information for the "meta": field
- will raise an EECCException if FileToSign does not exist
- returns the .sign file name, which is in fact FileToSign+.sign'
- use TECCSignatureCertifiedFile class to load and validate such files

function SignToBase64(Data: pointer; Len: integer): RawUTF8; overload;

Compute a base-64 encoded signature of some digital content
- memory buffer will be hashed using SHA-256, then will be signed using ECDSA over the private
  secret key of this certificate instance
- you could later on verify this text signature according to the public key of this certificate, calling
  TECCCertificateChain.IsSigned()
- create internally a temporary TECCSignatureCertified instance

function SignToBase64(const Hash: THash256): RawUTF8; overload;

Compute a base-64 encoded signature of some digital content hash
- signature will be certified by private secret key of this instance
- you could later on verify this text signature according to the public key of this certificate, calling
  TECCCertificateChain.IsSigned()
- supplied hash is likely to be from SHA-256, but could be e.g. crc256c
- create internally a temporary TECCSignatureCertified instance

property AFSplitStripes: integer read fAFSplitStripes;

How many anti-forensic diffusion stripes are used for private key storage
- default is 0, meaning no diffusion, i.e. 32 bytes of storage space
- you may set e.g. to 32 to activate safe diffusion to 1KB of storage for ToBase64/SaveToStream
  methods
- is modified temporarily by SaveToSecure() method

property StoreOnlyPublicKey: boolean read fStoreOnlyPublicKey write
fStoreOnlyPublicKey;

Disable private secret key storage in SaveToStream()
- default is false, i.e. the private secret key will be serialized
- you may set TRUE here so that SaveToStream() would store only the public certificate, as
  expected by a TECCCertificate class
- is used e.g. by PublicToBase64 method to trim the private information

TECCCertificateSecretSetting = class(TSynPersistentWithPassword)

Store settings pointing to a local .private file containing a secret key
- following TECCCertificateSecret secure binary file format
- you may use "ECC infocrypt" command to retrieve SaveToSource constants

constructor Create; override;

Initialize the settings with default values
function CertificateSecret(const FolderName: TFileName): TECCCertificateSecret;

Generate a TECCCertificateSecret instance corresponding to the settings
- is a wrapper around TECCCertificateSecret.CreateFromFile
- will read the FileName file (if supplied), or search for the <Serial>.private file in the supplied
  folder otherwise, then use associated Password/PasswordRounds values to uncypher it
- returns nil if Serial and FileName are '', or raise an exception on unexpected error
- caller is responsible of freeing the returned class instance

generate TECCCertificateSecret C

property FileName: TFileName read fFileName write fFileName;

The first characters of the .private file holding the secret key
- equals "" by default, meaning no private secret is defined
- you may use the Serial property instead to search in an application specific folder

property Password: RawUTF8 read fPassword write fPassword;

The password used to protect the .private file
- matches the -authpass parameter used with "ECC decrypt" command, but with
  TSynPersistentWithPassword encryption
- i.e. matches ConstName_CYPH as generated by TECCCertificateSecret.SaveToSource

property PasswordRounds: integer read fPasswordRounds write fPasswordRounds;

Number of PBKDF2 rounds to be applied to the associated password
- matchesConstName_ROUNDS as generated by TECCCertificateSecret.SaveToSource
- matches the -authrounds parameter used with "ECC decrypt" command
- default is DEFAULT_ECCROUNDS, i.e. 60000

property Serial: RawUTF8 read fSerial write fSerial;

The first characters of the .private file holding the secret key
- equals "" by default, meaning no private secret is defined
- you may use the FileName property instead to specify a full path name

TECCCertificateDecryptSetting = class(TECCCertificateSecretSetting)

Store settings pointing to a local .private file containing a secret key for .synecc file decryption
- following TECCCertificateSecure binary file format
- publishes Salt and SaltRounds values, as expected by TECCCertificateSecret.Decrypt method

class TECCCertificateDecryptSetting

constructor Create; override;

Initialize the settings with default values

property Salt: RawUTF8 read fSalt write fSalt;

The Salt passphrase used to protect the .synecc encrypted file
- matches the -saltpass parameter used with "ECC crypt" command
- default is 'salt'

property SaltRounds: integer read fSaltRounds write fSaltRounds;

Number of PBKDF2 rounds to be applied to the associated Salt
- matches the -saltrounds parameter used with "ECC crypt" command
- default is DEFAULT_ECCROUNDS, i.e. 60000

TECCSignatureCertified = class(TSynPersistent)

A ECDSA secp256r1 digital signature of some content, signed by an authority
constructor Create; override;
  Initialize this signature

constructor CreateFrom(const binary: TECCSignatureCertifiedContent; NoException: boolean=false);
  Initialize this signature from a supplied binary
  - will raise an EECCException if the supplied binary content is incorrect

constructor CreateFromBase64(const base64: RawUTF8; NoException: boolean=false);
  Initialize this signature from a supplied base-64 encoded binary
  - will raise an EECCException if the supplied base64 is incorrect

constructor CreateFromFile(const signfilename: TFileName; NoException: boolean=false);
  Initialize this signature from the "sign": field of a JSON .sign file
  - will raise an EECCException if the supplied file is incorrect

constructor CreateNew(Authority: TECCCertificateSecret; Data: pointer; Len: integer); overload;
  Compute a new signature of some digital content
  - memory buffer will be hashed using SHA-256, then will be signed using ECDSA over the private secret key of the supplied Authority certificate

constructor CreateNew(Authority: TECCCertificateSecret; const Hash: THash256); overload;
  Compute a new signature of some digital content hash
  - supplied hash is likely to be from SHA-256, but could be e.g. crc256c
  - the hash will be signed using ECDSA over the private secret key of the supplied Authority certificate

function Check: boolean;
  Fast check of the binary buffer storage of this signature
  - performs basic checks, avoiding any void date, authority or signature
  - use Verify() or TECCCertificateChain.IsSigned() methods for full digital signature validation

function FromBase64(const base64: RawUTF8): boolean;
  Retrieve the signature from some base-64 encoded binary
  - returns true on success, false otherwise

function FromFile(const signfilename: TFileName): boolean; virtual;
  Retrieve the signature from the "sign": field of a JSON .sign file
  - returns true on success, false otherwise

function SaveToDERBinary: RawByteString;
  Save the ECDSA signature into a ASN.1's binary DER buffer
  - note that DER content only stores the ECDSA digital signature, so all certification information is lost

function SaveToDERFile(const FileName: TFileName): boolean;
  Save the ECDSA signature into a ASN.1's binary DER file
  - note that DER content only stores the ECDSA digital signature, so all certification information is lost - consider using TECCSignatureCertifiedFile instead
  - returns TRUE on success, FALSE otherwise
function ToBase64: RawUTF8;
    Persist the signature as some base-64 encoded binary

function ToVariant: variant; virtual;
    Returns a TDocVariant object of all published properties of this instance

function Verify(Authority: TECCCertificate; const hash: THash256): TECCValidity; overload;
    Check if this digital signature matches a given data hash
    - will check internal properties of the certificate (e.g. validity dates), and validate the stored
      ECDSA signature according to the public key of the supplied signing authority
    - supplied hash is likely to be from SHA-256, but could be e.g. crc256c
    - this method is thread-safe, and not blocking

function Verify(Authority: TECCCertificate; Data: pointer; Len: integer): TECCValidity; overload;
    Check if this digital signature matches a given memory buffer
    - will check internal properties of the certificate (e.g. validity dates), and validate the stored
      ECDSA signature according to the public key of the supplied signing authority
    - will compute and verify the SHA-256 hash of the supplied data
    - this method is thread-safe, and not blocking

property AuthorityIssuer: RawUTF8 read GetAuthorityIssuer;
    Identify the authority issuer used for signing, as text

property AuthoritySerial: RawUTF8 read GetAuthoritySerial;
    Hexadecimal text of the authority certificate identifier used for signing

property Content: TECCSignatureCertifiedContent read fContent write fContent;
    Low-level access to the binary buffer used ECDSA secp256r1 cryptography
    - you should not use this property, but other methods

property Date: RawUTF8 read GetDate;
    When this signature was generated, as ISO-8601 text

property Version: word read fContent.Version;
    The TECCSignatureCertified format version
    - currently equals 1

TECCSignatureCertifiedFile = class(TECCSignatureCertified)
Handle a .sign file content as generated by TECCCertificateSecret.SignFile
    - JSON document of a SHA-256/ECDSA secp256r1 digital signature

constructor CreateFromDecryptedFile(const aDecryptedContent: RawByteString; const
    Signature: TECCSignatureCertifiedContent; const MetaData: RawJSON);
Create and set .sign fields after TECCCertificateSecret.Decrypt() process
    - will compute Size, and MD5/SHA-256 hashes from aDecryptedContent
    - will raise an EECCException if the supplied parameters are incorrect

function FromDecryptedFile(const aDecryptedContent: RawByteString; const
    Signature: TECCSignatureCertifiedContent; const MetaData: RawJSON): boolean;
Compute .sign fields after TECCCertificateSecret.Decrypt() process
    - will compute Size, and MD5/SHA-256 hashes from aDecryptedContent
**function** FromFile(const aFileName: TFileName): boolean; override;

*Read a .sign digital signature file*
- as previously generated by TECCCertificateSecret.SignFile
- will append '.sign' to aFileName, if it does not match this extension
- returns true on success, false otherwise

**function** FromFileJson(const aFileContent: RawUTF8): boolean;

*Read a .sign digital signature JSON content*
- as previously generated by TECCCertificateSecret.SignFile
- returns true on success, false otherwise

**property** LowLevelInfo: TDocVariantData read fLowLevelInfo;

*Low-level access to the whole JSON document members*

**property** MD5: RawUTF8 read fMD5;

*The MD5 hexadecimal signature as stored in the .sign file*

**property** MD5Digest: TMD5Digest read fMD5Digest;

*The MD5 binary hash as stored in the .sign file*

**property** MetaData: variant read fMetaData;

*The meta data document as stored in the .sign file*

**property** SHA256: RawUTF8 read fSHA256;

*The SHA-256 hexadecimal signature as stored in the .sign file*

**property** SHA256Digest: TSHA256Digest read fSHA256Digest;

*The SHA-256 binary hash as stored in the .sign file*

**property** Size: integer read fSize;

*The signed file size in bytes, as stored in the .sign file*

**TECCCertificateChain** = class(TSynPersistentLock)

*Manage PKI certificates using ECC secp256r1 cryptography*
- will implement a simple and efficient public-key infrastructure (PKI), based on JSON objects or even plain base-64 encoded JSON strings
- consider using TECCCertificateChainFile from mORMot.pas if you want to use convenient human-readable JSON serialization in files

**constructor** CreateFromArray(const values: TRawUTF8DynArray);

*Initialize the certificate store from an array of base-64 encoded strings*
- a TRawUTF8DynArray value is very convenient when storing the certificates chain as part of JSON settings, e.g. TDDDAppSettings
- will call LoadFromArray(), and raise EECCException on any error
constructor CreateFromFile(const jsonfile: TFileName);
  *Initialize the certificate store from some JSON-serialized .ca file*
  
  - the file would store plain verbose information of all certificates, i.e. base-64 full information (containing only public keys) and also high-level published properties of all stored certificates (e.g. Serial)
  
  - as such, this file format is more verbose than CreateFromJson/SaveToJson and may be convenient for managing certificates with a text/json editor
  
  - you may use SaveToFile() method to create such JSON file
  
  - will call LoadFromFile(), and raise EECCException on any error

constructor CreateFromFiles(const files: array of TFileName);
  *Initialize the certificate store from an array of .public file names*
  
  - raise EECCException on any error when reading a .public file

constructor CreateFromJson(const json: RawUTF8);
  *Initialize the certificate store from some JSON array of strings*
  
  - the serialization format is just a JSON array of base-64 encoded certificates (with only public keys) - so diverse from CreateFromFile()
  
  - will call LoadFromJson(), and raise EECCException on any error

destructor Destroy; override;
  *Finalize the certificate store*

function Add(cert: TECCCertificate): integer;
  *Register a certificate in the internal certificate chain*
  
  - returns the index of the newly inserted certificate
  
  - returns -1 on error, e.g. if the certificate was not valid, or its serial was already part of the internal list
  
  - any self-signed certificate will be rejected: use AddSelfSigned() instead
  
  - this method is thread-safe

function AddSelfSigned(cert: TECCCertificate): integer;
  *Register a self-signed certificate in the internal certificate chain*
  
  - a self-signed certificate will have its AuthoritySerial/AuthorityIssuer fields matching Serial/Issuer, and should be used as "root" certificates
  
  - returns -1 on error, e.g. if the certificate was not valid, not self-signed or its serial was already part of the internal list
  
  - this method is thread-safe

function GetBySerial(const Serial: TECCCertificateID): TECCCertificate; overload;
  *Search for a certificate from its binary identifier*
  
  - this method is not thread-safe, unless you use Safe.Lock/Unlock

function GetBySerial(const Serial: RawUTF8): TECCCertificate; overload;
  *Search for a certificate from its hexadecimal text identifier*
  
  - this method is not thread-safe, unless you use Safe.Lock/Unlock

function GetBySerial(const Serial: TECCCertificateID; out PublicKey: TECCPublicKey): boolean; overload;
  *Search for a certificate public key from its binary identifier*
  
  - returns TRUE if the Serial identifier was found, FALSE otherwise
  
  - this method is thread-safe, since it will make a private copy of the key
function GetBySerial(const Serial: TECCCertificateID; out Content: TECCCertificateContent): boolean; overload;

*Search for a certificate binary content from its binary identifier*
- returns TRUE if the Serial identifier was found, FALSE otherwise
- this method is thread-safe, since it will make a private copy of the content

function IsAuthorized(const base64sign: RawUTF8): boolean; overload;

*Check if the digital signature is recognized by the stored certificates*
- will check that the supplied base64 encoded text is a ECC signature, and that its AuthoritySerial is part of the Items[] list
- this method won't perform the ECDSA verification: use IsSigned() instead
- this method is thread-safe, and not blocking

function IsAuthorized(const sign: TECCSignatureCertifiedContent): boolean; overload;

*Check if the digital signature is recognized by the stored certificates*
- will check that sign.AuthoritySerial is part of the Items[] list
- this method won't perform the ECDSA verification: use IsSigned() instead
- this method is thread-safe, and not blocking

function IsAuthorized(sign: TECCSignatureCertified): boolean; overload;

*Check if the digital signature is recognized by the stored certificates*
- will check that sign.AuthoritySerial is part of the Items[] list
- this method won't perform the ECDSA verification: use IsSigned() instead
- this method is thread-safe, and not blocking

function IsSigned(const sign: TECCSignatureCertifiedContent; Data: pointer; Len: integer): TECCValidity; overload;

*Check if the digital signature of a given memory buffer is valid*
- will check internal properties of the certificate (e.g. validity dates), and validate the stored ECDSA signature according to the public key of the associated signing authority (which should be stored in Items[])
- will compute and verify the SHA-256 hash of the supplied data
- this method is thread-safe, and not blocking

function IsSigned(const base64sign: RawUTF8; const hash: THash256): TECCValidity;

*Verify the base-64 encoded digital signature of a given hash*
- will check internal properties of the certificate (e.g. validity dates), and validate the stored ECDSA signature according to the public key of the associated signing authority (which should be stored in Items[])
- supplied hash is likely to be from SHA-256, but could be e.g. crc256c
- this method is thread-safe, and not blocking

function IsSigned(const base64sign: RawUTF8; Data: pointer; Len: integer): TECCValidity; overload;

*Verify the base-64 encoded digital signature of a given memory buffer*
- will check internal properties of the certificate (e.g. validity dates), and validate the stored ECDSA signature according to the public key of the associated signing authority (which should be stored in Items[])
- will compute and verify the SHA-256 hash of the supplied data
- this method is thread-safe, and not blocking
function IsSigned(const sign: TECCSignatureCertifiedContent; const hash: THash256): TECCValidity; overload;

*Check if the digital signature of a given data hash is valid*
- will check internal properties of the certificate (e.g. validity dates), and validate the stored ECDSA signature according to the public key of the associated signing authority (which should be stored in Items[])
- supplied hash is likely to be from SHA-256, but could be e.g. crc256c
- this method is thread-safe, and not blocking

function IsSigned(sign: TECCSignatureCertified; const hash: THash256): TECCValidity; overload;

*Check if the digital signature of a given data hash is valid*
- will check internal properties of the certificate (e.g. validity dates), and validate the stored ECDSA signature according to the public key of the associated signing authority (which should be stored in Items[])
- supplied hash is likely to be from SHA-256, but could be e.g. crc256c
- this method is thread-safe, and not blocking

function IsSigned(sign: TECCSignatureCertified; Data: pointer; Len: integer): TECCValidity; overload;

*Check if the digital signature of a given memory buffer is valid*
- if sign is a TECCSignatureCertifiedFile, the Size, MD5 and SHA256 fields stored in the .sign file content will be checked against the supplied data before ECDSA signature, and would return ecvCorrupted on error
- it will then check internal properties of the certificate (e.g. validity dates), and validate the stored SHA-256/ECDSA signature according to the public key of the associated signing authority (stored in Items[])
- this method is thread-safe, and not blocking

function IsSigned(sign: TECCSignatureCertifiedFile): TECCValidity; overload;

*Check if the digital signature file (.sign content) is valid*
- will check internal properties of the certificate (e.g. validity dates), and validate the stored ECDSA signature according to the public key of the associated signing authority (which should be stored in Items[])
- will use TECCSignatureCertifiedFile Size, MD5 and SHA256 fields, so could be used without any actual memory buffer
- this method is thread-safe, and not blocking

function IsValid(const content: TECCCertificateContent; ignoreDate: boolean=false): TECCValidity; overload;

*Check if the certificate is valid, against known certificates chain*
- will check internal properties of the certificate (e.g. validity dates, unless ignoreDate=TRUE), and validate the stored ECDSA signature according to the public key of the associated signing authority (which should be valid, and stored in Items[])
- consider setting IsValidCached property to TRUE to reduce resource use
- this method is thread-safe, and not blocking
**isValid(cert: TECCCertificate): TECCValidity; overload;**

- Check if the certificate is valid, against known certificates chain
- will check internal properties of the certificate (e.g. validity dates), and validate the stored ECDSA signature according to the public key of the associated signing authority (which should be stored in Items[])
- consider setting IsValidCached property to TRUE to reduce resource use
- this method is thread-safe, and not blocking

**function LoadFromArray(const values: TRawUTF8DynArray): boolean;**

- Load a certificates chain from an array of base-64 encoded content
- would create only TECCCertificate instances with their public keys, since no private key, therefore no TECCCertificateSecret is expected

**function LoadFromFile(const jsonfile: TFileName): boolean;**

- Load a certificates chain from some JSON-serialized .ca file
- you may use SaveToFile() method to create such JSON file
- would create only TECCCertificate instances with their public keys, since no private key, therefore no TECCCertificateSecret is expected
- if jsonfile is not in the current folder, will try ECCKeyFileFolder

**function LoadFromFileContent(const cajsoncontent: RawUTF8): boolean;**

- Load a certificates chain from some JSON-serialized .ca file content
- you may use SaveToFileContent method to create such JSON content
- would create only TECCCertificate instances with their public keys, since no private key, therefore no TECCCertificateSecret is expected

**function LoadFromJson(const json: RawUTF8): boolean;**

- Load a certificates chain from a JSON array of strings
- follows SaveToJson format, i.e. base-64 encoded strings
- would create only TECCCertificate instances with their public keys, since no private key, therefore no TECCCertificateSecret is expected

**function SaveToArray: TRawUTF8DynArray;**

- Save the whole certificates chain as an array of base-64 encoded content
- each certificate would be stored via PublicToBase64() into a RawUTF8
- any private key would be trimmed from the output: private secret keys should NOT be kept in the main chain, in which only public keys will appear

**function SaveToFile(const jsonfile: TFileName): boolean;**

- Save the whole certificates chain as a .ca JSON file
- is in fact the human-friendly JSON serialization of this instance
- the .ca file would store plain verbose information of all certificates, i.e. base-64 full information (containing only public keys) and also high-level published properties of all stored certificates (e.g. Serial)
- as such, this file format is more verbose than CreateFromJson/SaveToJson and may be convenient for managing certificates with a text/json editor
function SaveToFileContent: RawUTF8;

Save the whole certificates chain as a JSON content, matching .ca format
- is in fact the human-friendly JSON serialization of this instance
- would store plain verbose information of all certificates, i.e. base-64 full information
  (containing only public keys) and also high-level published properties of all stored certificates
  (e.g. Serial)
- as such, .ca file format is more verbose than CreateFromJson/SaveToJson and may be
  convenient for managing certificates with a text/json editor

function SaveToFileVariant: variant;

Save the whole certificates chain as a JSON object, matching .ca format
- is in fact the human-friendly JSON serialization of this instance
- would store plain verbose information of all certificates, i.e. base-64 full information
  (containing only public keys) and also high-level published properties of all stored certificates
  (e.g. Serial)
- as such, .ca file format is more verbose than CreateFromJson/SaveToJson and may be
  convenient for managing certificates with a text/json editor

function SaveToJson: RawUTF8;

Save the whole certificates chain as a JSON array
- each certificate would be stored via PublicToBase64() into a JSON string
- any private key would be trimmed from the output JSON: private secret keys should NOT be
  kept in the main chain, in which only public keys should appear

function ValidateItems: TECCCertificateObjArray;

Check all stored certificates and their authorization chain
- returns nil if all items were valid
- returns the list of any invalid instances
- do not free the returned items, since they are reference to Items[]

procedure Clear;

Delete all stored certificates
- this method is thread-safe, calling Safe.Lock/Unlock

property Count: integer read GetCount;

How many certificates are currently stored in the certificates chain

property IsValidCached: boolean read fIsValidCached write SetIsValidCached;

If the IsValid() calls should maintain a cache of all valid certificates
- will use a naive but very efficient crc64c hashing of previous contents
- since ecdsa_verify() is very demanding, such a cache may have a huge speed benefit if the
  certificates are about to be supplied several times
- is disabled by default, for paranoid safety

property Items: TECCCertificateObjArray read fItems;

Low-level access to the internal certificates chain
- thread-safe process may be done using
  Safe.Lock; try ... finally Safe.Unlock; end;
TECDHEProtocol = class(TInterfacedObjectLocked)

Abstract ECDHE secure protocol with unilateral or mutual authentication
- inherited TECDHEProtocolClient and TECDHEProtocolServer classes will implement a secure client/server transmission, with a one-way handshake and asymmetric encryption
- will validate ECDSA signatures using certificates of the associated PKI
- will create an ephemeral ECC key pair for perfect forward security
- will use ECDH to compute a shared ephemeral session on both sides, for AES-128 or AES-256 encryption, and HMAC with anti-replay - default algorithm will use fast and safe AES-CFB 128-bit encryption, with efficient AES-CRC 256-bit MAC, and full hardware acceleration on Intel CPUs

constructor Create(aAuth: TECDHEAuth; aPKI: TECCCertificateChain; aPrivate: TECCCertificateSecret; reintroduce; overload; virtual;

Initialize the ECDHE protocol with a PKI and a private secret key
- if aPKI is not set, the certificates won't be validated and the protocol will allow self-signed credentials
- aPrivate should always be set for mutual or unilateral authentication
- will implement unilateral authentication if aPrivate=nil for this end

constructor CreateFrom(aAnother: TECDHEProtocol); virtual;
Will create another instance of this communication protocol

destructor Destroy; override;

Finalize the instance
- also erase all temporary secret keys, for safety

function CheckError(const aEncrypted: RawByteString): TProtocolResult; virtual;
Check for any transmission error of the supplied encrypted text
- returns sprSuccess if the stored CRC of the encrypted flow matches
- returns sprInvalidMAC in case of wrong aEncrypted input
- is only implemented for MAC=macDuringEF, otherwise returns sprUnsupported
- to be called before Decrypt(), since this later method will change the internal kM[false] sequence number

function Clone: IProtocol;
Will create another instance of this communication protocol

function Decrypt(const aEncrypted: RawByteString; out aPlain: RawByteString): TProtocolResult; virtual;

Decrypt a message on one side, as transmitted from the other side
- will use the Encryption Function EF, according to the shared secret key
- returns sprInvalidMAC in case of wrong aEncrypted input (e.g. packet corruption, MiM or Replay attacks attempts)
- this method is thread-safe
class function FromKey(const aKey: RawUTF8; aServer: boolean): TECDEHProtocol;

Creates a new TECDEHProtocolClient or TECDEHProtocolServer from a text key
- expected layout is values separated by ; with at least a=... pair
- if needed, you can specify p=... as the password file name (searching for first matching unique file name with .private extension in the current directory of in ECCKeyFileFolder), and pw=...;pr=... for the associated password protection (password content and rounds)
- optional ca=..;a=..;k=..;e=..;m=.. switches will match PKI, Auth, KDF, EF and MAC properties of this class instance (triming left lowercase chars)
- global value set by FromKeySetCA() is used as PKI, unless ca=.. is set (as a .ca file name, or as ca=base64,base64 or ca=base64","base64")
- a full text key with default values may be:
  a=mutual;k=hmacsha256;e=aescrc128;m=ef; p=34a2;pw=passwordFor34a2; pr=60000;ca=websockets
- returns nil if aKey does not match this format, i.e. has no p=...,pw=..

class function FromKeyCompute(const privkey, privpassword: RawUTF8; privrounds: integer=DEFAULT_ECCROUNDS; const pki: RawUTF8=''; auth: TECDEHEAuth=authMutual; kdf: TECDEHKDF=kdfHmacSha256; ef: TECDEHEEF=efAesCrc128; mac: TECDEHEMAC=macDuringEF; customkey: cardinal=0): RawUTF8;

Computes a TSynPersistentWithPassword key expected by FromKey
- the .private key file name, and its associated password/rounds should be specified, but for unilateral authentication on the other side
- pki should be a .ca file name, 'base64,base64' or '"base64","base64"
- result of this method can be stored directly in a .settings file, to enable the TECDEHProtocol safe protocol for transmission

function ProcessHandshake(const MsgIn: RawUTF8; out MsgOut: RawUTF8): TProtocolResult; virtual; abstract;

Initialize the communication by exchanging some client/server information
- this method should be overriden with the proper implementation

procedure Encrypt(const aPlain: RawByteString; out aEncrypted: RawByteString); virtual;

Encrypt a message on one side, ready to be transmitted to the other side
- will use the Encryption Function EF, according to the shared secret key
- this method is thread-safe

class procedure FromKeySetCA(aPKI: TECCCertificateChain);

Defines the default PKI instance to be used by FromKey
- used if the ca=... property is not set in the aKey value

property Auth: TECDEHEAuth read fAlgo.auth;

The current Authentication scheme
- this value on client side should match server's Authorized
- this value on server side may change if the client forced another mode

property CertificateValidity: TECCValidity read fCertificateValidity;

After handshake, contains the information about the other side public key certificate validity, against the shared PKI

property EF: TECDEHEEF read fAlgo.ef write fAlgo.ef;

The current Encryption Function
- this value should match on both client and server sides
property EFSalt: RawByteString read fEFSalt write fEFSalt;

The current salt, used by the Key Derivation Function KDF to compute the key supplied to the Encryption Function EF
- equals 'ecdhesalt' by default
- this value should match on both client and server sides

property KDF: TECDEHekDF read fAlgo.kdf write fAlgo.kdf;

The current Key Derivation Function
- this value should match on both client and server sides

property MAC: TECDEHMAC read fAlgo.mac write fAlgo.mac;

The current Message Authentication Code
- this value should match on both client and server sides

property MACSalt: RawByteString read fMACSalt write fMACSalt;

The current salt, used by the Key Derivation Function KDF to compute the key supplied to the Message Authentication Code MAC
- equals 'ecdhemac' by default
- this value should match on both client and server sides

property PKI: TECCertificateChain read fPKI;

Shared public-key infrastructure, used to validate exchanged certificates
- will be used for authenticity validation of ECDSA signatures

TECDHEProtocolClient = class(TECDHEProtocol)
Implements ECDHE secure protocol on client side

constructor Create(aAuth: TECDEHAuth; aPKI: TECCertificateChain; aPrivate: TECCertificateSecret); override;

Initialize the ECDHE protocol on the client side
- will check that aAuth is compatible with the supplied aPKI/aPrivate

function ProcessHandshake(const MsgIn: RawUTF8; out MsgOut: RawUTF8): TProtocolResult; override;

Initialize the client communication
- if MsgIn is ", will call ComputeHandshake
- if MsgIn is set, will call ValidateHandshake

function ValidateHandshake(const aServer: TECDEHFrameServer): TProtocolResult;

Validate the authentication frame sent back by the server

procedure ComputeHandshake(out aClient: TECDEHFrameClient);

Generate the authentication frame sent from the client

TECDHEProtocolServer = class(TECDHEProtocol)
Implements ECDHE secure protocol on server side

constructor Create(aAuth: TECDEHAuth; aPKI: TECCertificateChain; aPrivate: TECCertificateSecret); override;

Initialize the ECDHE protocol on the client side
- will check that aAuth is compatible with the supplied aPKI/aPrivate
constructor CreateFrom(aAnother: TECDEHProtocol); override;
Will create another instance of this communication protocol

function ComputeHandshake(const aClient: TECDEHFrameClient; out aServer: TECDEHFrameServer): TProtocolResult;
Generate the authentication frame corresponding to the client request
- may change Auth property if the Client requested another authentication scheme, allowed in Authorized setting and compatible with fPrivate

function ProcessHandshake(const MsgIn: RawUTF8; out MsgOut: RawUTF8): TProtocolResult; override;
Initialize the server communication
- will call ComputeHandshake

property Authorized: TECDEHAuths read fAuthorized write fAuthorized;
The Authentication Schemes allowed by this server
- by default, only the aAuth value specified to Create is allowed
- you can set e.g. [authMutual,authServer] for a weaker pattern

TJWTES256 = class(TJWTAbstract)
Implements JSON Web Tokens using ’ES256’ algorithm
- i.e. ECDSA using the P-256 curve and the SHA-256 hash algorithm
- as defined in http://tools.ietf.org/html/rfc7518 paragraph 3.4
- since ECDSA signature and verification is CPU consumming (under x86, it takes 2.5 ms, but only 0.3 ms on x64) you may enable CacheTimeoutSeconds

constructor Create(aCertificate: TECCCertificate; aClaims: TJWTClaims; const aAudience: array of RawUTF8; aExpirationMinutes: integer=0; aIDIdentifier: TSynUniqueIdentifierProcess=0; aIDObfuscationKey: RawUTF8=''); reintroduce;
Initialize the JWT processing instance using ECDSA P-256 algorithm
- the supplied set of claims are expected to be defined in the JWT payload
- the supplied ECC certificate should be a TECCCertificate storing the public key needed for Verify(), or a TECCCertificateSecret storing also the private key required by Compute()
- aCertificate is owned by this instance if property OwnCertificate is true
- aAudience are the allowed values for the jrcAudience claim
- aExpirationMinutes is the deprecation time for the jrcExpirationTime claim
- aIDIdentifier and aIDObfuscationKey are passed to a TSynUniqueIdentifierGenerator instance used for jrcJwtID claim

destructor Destroy; override;
Finalize the instance

property Certificate: TECCCertificate read fCertificate;
Access to the associated TECCCertificate instance
- which may be a TECCCertificateSecret for Compute() private key

property OwnCertificate: boolean read fOwnCertificate write fOwnCertificate;
If the associated TECCCertificate is to be owned by this instance

Types implemented in the SynEcc unit
PECCCertificateContent = ^TECCCertificateContent;
Points to a TECCCertificate binary buffer for ECC secp256r1 cryptography

PECCCertificateSigned = ^TECCCertificateSigned;
Points to certification information of a TECCCertificate

PECSSignatureCertifiedContent = ^TECSSignatureCertifiedContent;
Points to a TECSSignatureCertified buffer for ECDSA secp256r1 signature

PECDEA1go = ^TECDHEAlgorithm;
Points to one protocol Algorithm recognized by TECDHEProtocol

PECIESHeader = ^TECIESHeader;
Points to the binary header of a .synecc encrypted file

TECCCertificateID = type THash128;
Used to identify a TECCCertificate
- could be generated by TAESPRNG.Fill() method

TECCCertificateIssuer = type THash128;
Used to identify a TECCCertificate issuer
- could be generated by AsciiToBaudot(), with truncation to 16 bytes (up to 25 Ascii-7 characters)

TECCCertificateObjArray = array of TECCCertificate;
Used to store a list of TECCCertificate instances
- e.g. in TECCCertificateChain.Items
- TJSONSerializer.RegisterObjArrayForJSON done in dddInfraApps and not in this unit to avoid dependency to mORMot.pas

TECCDate = word;
Used to store a date in a TECCCertificate
- i.e. 16-bit number of days since 1 August 2016
- use NowECCDate, ECCDate(), ECCToDateTime() or ECCText() functions

TECCDecrypt = ( ecdDecrypted, ecdDecryptedWithSignature, ecdNoContent, ecdCorrupted, ecdInvalidSerial, ecdNoPrivateKey, ecdInvalidMAC, ecdDecryptError, ecdWriteFileError );
The error codes returned by TECCCertificateSecret.Decrypt()
- see also ECC_VALIDDECYPRET constant

TECCHash = THash256;
Store a 256-bit hash, as expected by ECC secp256r1 cryptography
- see e.g. ecda_sign() and ecda_verify() functions

TECCPrivateKey = array[0..ECC_BYTES-1] of byte;
Store a private key for ECC secp256r1 cryptography
- use ecc_make_key() to generate such a key
- stored in compressed form, i.e. each private key consumes 32 bytes of memory

TECCPublicKey = array[0..ECC_BYTES] of byte;
Store a public key for ECC secp256r1 cryptography
- use ecc_make_key() to generate such a key
- stored in compressed form with its standard byte header, i.e. each public key consumes 33 bytes of memory

TECCPublicKeyUncompressed = array[0..(ECC_BYTES*2)-1] of byte;
Store a public key for ECC secp256r1 cryptography
- use ecc_uncompress_key_pasn() to compute such a key from a TECCPublicKey
- stored in uncompressed form, consuming 64 bytes of memory

\[ \text{TECCSecretKey} = \text{THash256}; \]

Store an encryption key, as generated by ECC secp256r1 cryptography
- use ecdh_shared_secret() to compute such a key from public/private keys
- 256-bit / 32 bytes derivation from secp256r1 ECDH is expected to have at least 247 bits of entropy
so could better be derivated via a KDF before used as encryption secret - see @http://crypto.stackexchange.com/a/9428/40200

\[ \text{TECCSignature} = \text{array[0..(ECC_BYTES*2)-1]} \]

Store a signature, as generated by ECC secp256r1 cryptography
- see e.g. ecdsa_sign() and ecdsa_verify() functions
- contains ECDSA's R and S integers
- each ECC signature consumes 64 bytes of memory

\[ \text{TECCValidity} = ( \text{ecvUnknown, ecvValidSigned, ecvValidSelfSigned, ecvNotSupported,} \]
\[ \text{ecvBadParameter, ecvCorrupted, ecvInvalidDate, ecvUnknownAuthority,} \]
\[ \text{ecvDeprecatedAuthority, ecvInvalidSignature}; \]

\[ \text{Indicate the validity state of a ECDSA signature against a certificate} \]
- as returned by low-level ECCVerify() function, and TECCSignatureCertified.Verify, TECCCertificateChain.IsValid or TECCCertificateChain.IsSigned methods
- see also ECC_VALIDSIGN constant

\[ \text{TECDHEAuth} = ( \text{authMutual, authServer, authClient}); \]

\[ \text{The Authentication schemes recognized by TECDHEProtocol} \]
- specifying the authentication allows a safe one-way handshake

\[ \text{TECDHEAuths} = \text{set of TECDHEAuth}; \]

\[ \text{Set of Authentication schemes recognized by TECDHEProtocolServer} \]

\[ \text{TECDHEEF} = ( \text{efAesCrc128, efAesCfb128, efAesOfb128, efAesCtr128, efAesCbc128,} \]
\[ \text{efAesCrc256, efAesCfb256, efAesOfb256, efAesCtr256, efAesCbc256 }); \]

\[ \text{The Encryption Functions recognized by TECDHEProtocol} \]
- all supported AES chaining blocks have their 128-bit and 256-bit flavours
- default efAesCrc128 will use the dedicated TAECSFBBCRC class, i.e. AES-CFB encryption with
on-the-fly 256-bit CRC computation of the plain and encrypted blocks, and AES-encryption of the
CRC to ensure cryptographic level message authentication and integrity - associated TECDEMAC
property should be macDuringEF
- other values will define TAECSFB/TAESOFB/TAECSCTR/TAESCBC in 128-bit or 256-bit mode, in
conjunction with a TECDEMAC setting
- AES-NI hardware acceleration will be used, if available - under x86-64, efAesOfb128 will potentially
give the best performance
- of course, weack ECB mode is not available

\[ \text{TECDHEKDF} = ( \text{kdfHmacSha256}); \]

\[ \text{The Key Derivation Functions recognized by TECDHEProtocol} \]
- used to compute the EF secret and MAC secret from shared ephemeral secret
- only HMAC SHA-256 safe algorithm is proposed currently

\[ \text{TECDHMAC} = ( \text{macDuringEF, macHmacSha256, macHmacCrc256c, macHmacCrc32c, macXxHash32,} \]
\[ \text{macNone}); \]

\[ \text{The Message Authentication Codes recognized by TECDHEProtocol} \]
- default macDuringEF (680MB/s for efAesCrc128 with SSE4.2 and AES-NI) means that no separated MAC is performed, but done during encryption step: only supported by efAesCrc128 or efAesCrc256 (may be a future AES-GCM)
- macHmacSha256 is the safest, but slow, especially when used as MAC for AES-NI accelerated encryption (110MB/s with efAesCfb128, to be compared with macDuringEF, which produces a similar level of MAC)
- macHmacCrc256c and macHmacCrc32c are faster (550-650MB/s with efAesCfb128), and prevent transmission errors but not message integrity or authentication since composition of two crcs is a multiplication by a polynomial - see http://mslc.ctf.su/wp/boston-key-party-ctf-2016-hmac-crc-crypto-5pts
- macXxHash32 will use the xxhash32() algorithm, fastest without SSE4.2
- macNone (800MB/s, which is the speed of AES-NI encryption itself for a random set of small messages) won't check errors, but only replay attacks

```pascal
TECDHEProtocol1Class = class of TECDEProtocol;
Meta-class of the TECDEProtocol type
```

```pascal
TECIESAlgo = ( ecaUnknown, ecaPBKDF2_HMAC_SHA256_AES256_CFB,
ecaPBKDF2_HMAC_SHA256_AES256_CBC, ecaPBKDF2_HMAC_SHA256_AES256_OFB,
ecaPBKDF2_HMAC_SHA256_AES256_CTR, ecaPBKDF2_HMAC_SHA256_AES256_CFB_SYNLZ,
ecaPBKDF2_HMAC_SHA256_AES256_CBC_SYNLZ, ecaPBKDF2_HMAC_SHA256_AES256_OFB_SYNLZ,
ecaPBKDF2_HMAC_SHA256_AES128_CFB_SYNLZ, ecaPBKDF2_HMAC_SHA256_AES128_CBC_SYNLZ,
ecaPBKDF2_HMAC_SHA256_AES128_OFB_SYNLZ, ecaPBKDF2_HMAC_SHA256_AES128_CTR,
ecaPBKDF2_HMAC_SHA256_AES128_CFB, ecaPBKDF2_HMAC_SHA256_AES128_CBC,
ecaPBKDF2_HMAC_SHA256_AES128_OFB, ecaPBKDF2_HMAC_SHA256_AES128_CTR );
```

The known algorithms implemented in ECIES encryption
- supports AES 256-bit encryption with safe block modes (weack ECB mode is not available) - or AES 128-bit if needed (e.g. for regulatory issues)
- safe HMAC SHA-256 is used as Message Authentication Code algorithm
- optional SynLZ compression can be enabled

Constants implemented in the SynEcc unit

```pascal
ECCCERTIFICATEPUBLIC_FILEEXT = '.public';
File extension of the JSON file storing a TECCCertificate public key

ECCCERTIFICATESECRET_FILEEXT = '.private';
File extension of the binary encrypted file storing a private key
- as generated by TECCCertificateSecret.SaveToSecureFile method

ECCCERTIFICATESIGN_FILEEXT = '.sign';
File extension of the JSON file storing a digital signature of a file
- by convention, this .sign extension is appended to the original file name
- as generated by TECCCertificateSecret.SignFile, and loaded by the TECCSignatureCertifiedFile class

ECCCERTIFICATES_FILEEXT = '.ca';
File extension of the JSON file storing a certificate authorities chain
- as generated by mORMot.pas TECCCertificateChainFile.SaveToFile() and loaded by TECCCertificateChain.LoadFromFile

ECC_BYTES = sizeof(THash256);
The size of the 256-bit memory structure used for secp256r1
- map 32 bytes of memory
**Function summaries**

**ECC_VALIDIDECRYPT** = [ecdDecrypted, ecdDecryptedWithSignature];

*TECCDecrypt results indicating a valid decryption process*

**ECC_VALIDISIGN** = [ecvValidSigned, ecvValidSelfSigned];

*TECCValidity results indicating a valid digital signature*

**ENCRYPTED_FILEEXT** = '.synecc';

*File extension of the ECIES encrypted file*
- with optional digital signature of the plain content
- as generated by TECCCertificate.Excel/EncryptFile, and decoded via TECCCertificateSecret.Decrypt

**Functions or procedures implemented in the SynEcc unit**

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECCCheck</td>
<td>Fast check of the binary buffer storage of a signature</td>
<td>1344</td>
</tr>
<tr>
<td>ECCCheckDate</td>
<td>Fast check of the dates stored in a certificate binary buffer</td>
<td>1344</td>
</tr>
<tr>
<td>ECDate</td>
<td>Convert a supplied TDateTime value into a TECCDate integer value</td>
<td>1344</td>
</tr>
<tr>
<td>ECCID</td>
<td>Convert a supplied hexadecimal buffer into a TECCCertificateID binary buffer</td>
<td>1344</td>
</tr>
<tr>
<td>ECCIssuer</td>
<td>Convert some Ascii-7 text into a TECCCertificateIssuer binary buffer</td>
<td>1344</td>
</tr>
<tr>
<td>ECCKeyFileFind</td>
<td>Search the single .public or .private file starting with the supplied file name</td>
<td>1344</td>
</tr>
<tr>
<td>ECCKeyFileFolder</td>
<td>Retrieve the private local folder used to store .public or .private files</td>
<td>1344</td>
</tr>
<tr>
<td>ECCSelfSigned</td>
<td>Fast check if the binary buffer storage of a certificate was self-signed</td>
<td>1344</td>
</tr>
<tr>
<td>ECCSign</td>
<td>Convert a supplied base-64 text into a TECCSignatureCertifiedContent binary buffer</td>
<td>1344</td>
</tr>
<tr>
<td>ECCText</td>
<td>Convert a supplied TECCCertificateCertifiedContent binary buffer into proper text</td>
<td>1345</td>
</tr>
<tr>
<td>ECCText</td>
<td>Convert a supplied TECCCertificateIssuer binary buffer into proper text</td>
<td>1345</td>
</tr>
<tr>
<td>ECCText</td>
<td>Convert a supplied TECCCertificate binary buffer into proper text</td>
<td>1345</td>
</tr>
<tr>
<td>ECCText</td>
<td>Convert a supplied TECCDate integer value into a ISO-8601 text value</td>
<td>1345</td>
</tr>
<tr>
<td>ECCText</td>
<td>Convert a supplied TECCCertificateID binary buffer into proper text</td>
<td>1345</td>
</tr>
<tr>
<td>ECCToDateTime</td>
<td>Convert a supplied a TECCDate integer value into a TDateTime value</td>
<td>1345</td>
</tr>
<tr>
<td>ECCTest</td>
<td>Low-level verification of a TECCSignatureCertifiedContent binary buffer</td>
<td>1345</td>
</tr>
<tr>
<td>ecc_make_key</td>
<td>Create a public/private key pair</td>
<td>1345</td>
</tr>
<tr>
<td>ecc_make_key_pas</td>
<td>Pascal function to create a secp256r1 public/private key pair</td>
<td>1345</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>ecc_uncompress_keyPas</td>
<td>Uncompress a public key for ECC secp256r1 cryptography</td>
<td>1345</td>
</tr>
<tr>
<td>ecdh_shared_secret</td>
<td>Compute a shared secret given your secret key and someone else's public key</td>
<td>1346</td>
</tr>
<tr>
<td>ecdh_shared_secretPas</td>
<td>Pascal function to compute a secp256r1 shared secret given your secret key and someone else's public key (in compressed format)</td>
<td>1346</td>
</tr>
<tr>
<td>ecdh_shared_secretPas</td>
<td>Pascal function to compute a secp256r1 shared secret given your secret key and someone else's public key (in uncompressed/flat format)</td>
<td>1346</td>
</tr>
<tr>
<td>ecdsa_sign</td>
<td>Generate an ECDSA signature for a given hash value</td>
<td>1346</td>
</tr>
<tr>
<td>ecdsa_signPas</td>
<td>Pascal function to generate an ECDSA secp256r1 signature for a given hash value</td>
<td>1346</td>
</tr>
<tr>
<td>ecdsa_verify</td>
<td>Verify an ECDSA signature</td>
<td>1346</td>
</tr>
<tr>
<td>ecdsa_verifyPas</td>
<td>Pascal function to verify an ECDSA secp256r1 signature from someone else's public key (in uncompressed/flat format)</td>
<td>1347</td>
</tr>
<tr>
<td>ecdsa_verifyPas</td>
<td>Pascal function to verify an ECDSA secp256r1 signature from someone else's public key (in compressed format)</td>
<td>1347</td>
</tr>
<tr>
<td>ECIESHeader</td>
<td>Validate the binary header of a .synecc file buffer, encrypted via ECC secp256r1</td>
<td>1347</td>
</tr>
<tr>
<td>ECIESHeader</td>
<td>Extract the binary header of a .synecc file buffer, encrypted via ECC secp256r1</td>
<td>1347</td>
</tr>
<tr>
<td>ECIESHeaderFile</td>
<td>Extract the binary header of a .synecc file, encrypted via ECC secp256r1</td>
<td>1347</td>
</tr>
<tr>
<td>ECIESHeaderText</td>
<td>Convert the header of a .synecc file into a JSON object</td>
<td>1347</td>
</tr>
<tr>
<td>ECIESHeaderText</td>
<td>Convert the binary header of a .synecc file buffer into a JSON object</td>
<td>1347</td>
</tr>
<tr>
<td>ECIESKeyFileFind</td>
<td>Search the single .public or .private file used to crypt a given content</td>
<td>1347</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fill all bytes of this ECC private key buffer with zero</td>
<td>1347</td>
</tr>
<tr>
<td>IsEqual</td>
<td>Compare two TECCertificateIssuer binary buffer values</td>
<td>1348</td>
</tr>
<tr>
<td>IsEqual</td>
<td>Compare two TECCertificateID binary buffer values</td>
<td>1348</td>
</tr>
<tr>
<td>IsZero</td>
<td>Ensure a TECCertificateIssuer binary buffer is not void, i.e. filled with 0</td>
<td>1348</td>
</tr>
<tr>
<td>IsZero</td>
<td>Ensure a TECCertificateID binary buffer is not void, i.e. filled with 0</td>
<td>1348</td>
</tr>
<tr>
<td>NowECCDate</td>
<td>Returns the current UTC date, as a TECCDate integer value</td>
<td>1348</td>
</tr>
</tbody>
</table>


```pascal
function ECCCh(content: TECCCertificateContent): boolean; overload;

Fast check of the binary buffer storage of a certificate
- ensure content.CRC has the expected value, using FNV-1a checksum
- does not validate the certificate against the certificates chain, nor perform any ECC signature:
  use TECCCertificateChain.IsValid instead

function ECCCheck(content: TECCSignatureCertifiedContent): boolean; overload;

Fast check of the binary buffer storage of a signature
- just check that the date and authority are set

function ECCCheckDate(content: TECCCertificateContent): boolean;

Fast check of the dates stored in a certificate binary buffer
- could be validated against ECCCheck()

function ECCDate(DateTime: TDateTime): TECCDate;

Convert a supplied TDateTime value into a TECCDate integer value
- i.e. 16-bit number of days since 1 August 2016
- returns 0 if the supplied value is invalid, i.e. out of range

function ECCID(Text: RawUTF8; out ID: TECCCertificateID): boolean;

Convert a supplied hexadecimal buffer into a TECCCertificateID binary buffer
- returns TRUE if the supplied Text was a valid hexadecimal buffer

function ECCIssuer(Text: RawUTF8; out Issuer: TECCCertificateIssuer): boolean;

Convert some Ascii-7 text into a TECCCertificateIssuer binary buffer
- using Emile Baudot encoding
- returns TRUE on Text truncation to fit into the 16 bytes

function ECCKeyFileFind(var TruncatedFileName: TFileName; privkey: boolean): boolean;

Search the single .public or .private file starting with the supplied file name
- as used in the ECC.dpr command-line sample project
- returns true and set the full file name of the matching file
- returns false is there is no match, or more than one matching file
- will also search in ECCKeyFileFolder, if the supplied folder is not enough

function ECCKeyFileFolder: TFileName;

Retrieve the private local folder used to store .public or .private files
- it is better to store all you key files in a single place, for easier and safer management
- under Windows, returns 'C:\Users\username\AppData\Local\Synopse\Keys\'
- under Linux, returns '${HOME}/.synopse/keys/'

function ECCSelfSigned(content: TECCCertificateContent): boolean;

Fast check if the binary buffer storage of a certificate was self-signed
- a self-signed certificate will have its AuthoritySerial/AuthorityIssuer fields matching Serial/Issuer

function ECCSign(base64: RawUTF8; out content: TECCSignatureCertifiedContent): boolean;

Convert a supplied base-64 text into a TECCSignatureCertifiedContent binary buffer

function ECCText(ECCDate: TECCDate; Expanded: boolean=true): RawUTF8; overload;

Convert a supplied a TECCDate integer value into a ISO-8601 text value
- i.e. 16-bit number of days since 1 August 2016
```
function ECCText(const ID: TECCCertificateID): RawUTF8; overload;

Convert a supplied TECCCertificateID binary buffer into proper text
- returns hexadecimal values, or "$" if the ID is filled with zeros

function ECCText(const sign: TECCSignature): RawUTF8; overload;

Convert a supplied TECCSignature binary buffer into proper text
- returns base-64 encoded text, or "$" if the signature was filled with zeros

function ECCText(const sign: TECCSignatureCertifiedContent): RawUTF8; overload;

Convert a supplied TECCSignatureCertifiedContent binary buffer into proper text
- returns base-64 encoded text, or "$" if the signature was filled with zeros

function ECCText(const Issuer: TECCCertificateIssuer): RawUTF8; overload;

Convert a supplied TECCCertificateIssuer binary buffer into proper text
- returns Ascii-7 text if was stored using Baudot encoding
- or returns hexadecimal values, if it was 16 bytes of random binary

function ECCToDateTime(ECCDate: TECCDate): TDateTime;

Convert a supplied a TECCDate integer value into a TDateTime value
- i.e. 16-bit number of days since 1 August 2016

function ECCVerify(const sign: TECCSignatureCertifiedContent; const hash: THash256; const auth: TECCCertificateContent): TECCValidity;

Low-level verification of a TECCSignatureCertifiedContent binary buffer
- will verify all internal signature fields according to a supplied authority, then will perform the ECDSA verification of the supplied 256-bit hash with the authority public key
- as used by TECCSignatureCertified.Verify and TECCCertificateChain.IsValid

function ecc_make_key(out pub: TECCPublicKey; out priv: TECCPrivateKey): boolean;

cdecl;

Create a public/private key pair
- using secp256r1 curve, i.e. NIST P-256, or OpenSSL prime256v1
- directly low-level access to the statically linked easy-ecc library function
- returns true if the key pair was generated successfully in pub/priv
- returns false if an error occurred
- this function is thread-safe and does not perform any memory allocation

function ecc_make_key_pas(out PublicKey: TECCPublicKey; out PrivateKey: TECCPrivateKey): boolean;

Pascal function to create a secp256r1 public/private key pair
- used only on targets (e.g. ARM/PPC) when the static .o version is not available

procedure ecc_uncompress_key_pas(const Compressed: TECCPublicKey; out Uncompressed: TECCPublicKeyUncompressed);

Uncompress a public key for ECC secp256r1 cryptography
- convert from its compressed form with its standard byte header (33 bytes of memory) into uncompressed/flat form (64 bytes of memory)
function ecdh_shared_secret(const pub: TECCPublicKey; const priv: TECCPrivateKey; out secret: TECCSecretKey): boolean; cdecl;

Compute a shared secret given your secret key and someone else's public key
- using secp256r1 curve, i.e. NIST P-256, or OpenSSL prime256v1
- directly low-level access to the statically linked easy-ecc library function
- note: it is recommended that you hash the result of ecdh_shared_secret before using it for symmetric encryption or HMAC (via an intermediate KDF)
- returns true if the shared secret was generated successfully in secret
- returns false if an error occurred
- this function is thread-safe and does not perform any memory allocation

function ecdh_shared_secret_pas(const PublicPoint: TECCPublicKeyUncompressed; const PrivateKey: TECCPrivateKey; out Secret: TEccSecretKey): boolean; overload;
Pascal function to compute a secp256r1 shared secret given your secret key and someone else's public key (in uncompressed/flat format)
- this overloaded function is slightly faster than the one using TECCPublicKey, since public key doesn't need to be uncompressed

function ecdh_shared_secret_pas(const PublicKey: TECCPublicKey; const PrivateKey: TECCPrivateKey; out Secret: TECCSecretKey): boolean; overload;
Pascal function to compute a secp256r1 shared secret given your secret key and someone else's public key (in compressed format)
- used only on targets (e.g. ARM/PPC) when the static .o version is not available

function ecdsa_sign(const priv: TECCPrivateKey; const hash: TECCHash; out sign: TECCSignature): boolean; cdecl;

Generate an ECDSA signature for a given hash value
- using secp256r1 curve, i.e. NIST P-256, or OpenSSL prime256v1
- directly low-level access to the statically linked easy-ecc library function
- returns true if the signature generated successfully in sign
- returns false if an error occurred
- this function is thread-safe and does not perform any memory allocation

function ecdsa_sign_pas(const PrivateKey: TECCPrivateKey; const Hash: TECCHash; out Signature: TECCSignature): boolean;
Pascal function to generate an ECDSA secp256r1 signature for a given hash value
- used only on targets (e.g. ARM/PPC) when the static .o version is not available

function ecdsa_verify(const pub: TECCPublicKey; const hash: TECCHash; const sign: TECCSignature): boolean; cdecl;

Verify an ECDSA signature
- using secp256r1 curve, i.e. NIST P-256, or OpenSSL prime256v1
- directly low-level access to the statically linked easy-ecc library function
- returns true if the signature is valid
- returns false if an error occurred
- this function is thread-safe and does not perform any memory allocation

function ecdsa_verify_pas(const PublicKey: TECCPublicKey; const Hash: TECCHash; const Signature: TECCSignature): boolean; overload;
Pascal function to verify an ECDSA secp256r1 signature from someone else's public key (in compressed format)
- used only on targets (e.g. ARM/PPC) when the static .o version is not available
function ecdsa_verify_pas(const PublicKey: TECCPublicKeyUncompressed; const Hash: TECCHash; const Signature: TECCSignature): boolean; overload;

Pascal function to verify an ECDSA secp256r1 signature from someone else's public key (in uncompressed/flat format)
- this overloaded function is slightly faster than the one using TECCPublicKey, since public key doesn't need to be uncompressed

function ECIESHeader(const encrypted: RawByteString; out head: TECIESHeader): boolean; overload;

Extract the binary header of a .synecc file buffer, encrypted via ECC secp256r1
- match the format generated by TECCCertificate.Encrypt/EncryptFile
- returns true on success, false otherwise

function ECIESHeader(const head: TECIESHeader): boolean; overload;

Validate the binary header of a .synecc file buffer, encrypted via ECC secp256r1
- will check against the expected layout, and values stored (e.g. crc)
- returns true if head is a valid .synecc header, false otherwise

function ECIESHeaderFile(const encryptedfile: TFileName; out head: TECIESHeader; const rawencryptedfile: TFileName=''): boolean;

Extract the binary header of a .synecc file, encrypted via ECC secp256r1
- match the format generated by TECCCertificate.Encrypt/EncryptFile
- returns true on success, false otherwise
- if rawencryptedfile is specified, will also create such a file with the raw encrypted content (i.e. excluding the encryptedfile header)

function ECIESHeaderText(const head: TECIESHeader): RawUTF8; overload;

Convert the binary header of a .synecc file buffer into a JSON object
- returns '' if the header is not a valid .synecc file

function ECIESHeaderText(const encryptedfile: TFileName; const rawencryptedfile: TFileName=''): RawUTF8; overload;

Convert the binary header of a .synecc file into a JSON object
- returns '' if the header is not a valid .synecc file
- if rawencryptedfile is specified, will also create such a file with the raw encrypted content (i.e. excluding the encryptedfile header)

function ECIESKeyFileFind(const encrypted: RawByteString; out keyfile: TFileName; privkey: boolean=true): boolean;

Search the single .public or .private file used to crypt a given content
- match the format generated by TECCCertificate.Encrypt/EncryptFile
- returns true on success, false otherwise
- will also search in ECCKeyFileFolder, if the current folder is not enough

procedure FillZero(out Priv: TECCPrivateKey); overload;

Fill all bytes of this ECC private key buffer with zero
- may be used to cleanup stack-allocated content
... finally FillZero(PrivateKey); end;

function IsEqual(const id1,id2: TECCCertificateID): boolean; overload;

Compare two TECCCertificateID binary buffer values
function IsEqual(const issuer1, issuer2: TECCertificateIssuer): boolean; overload;
    Compare two TECCertificateIssuer binary buffer values

function IsZero(const id: TECCertificateID): boolean; overload;
    Ensure a TECCertificateID binary buffer is not void, i.e. filled with 0

function IsZero(const issuer: TECCertificateIssuer): boolean; overload;
    Ensure a TECCertificateIssuer binary buffer is not void, i.e. filled with 0

function NowECCDate: TECCDate;
    Returns the current UTC date, as a TECCDate integer value
    - i.e. 16-bit number of days since 1 August 2016
27.20. SynFastWideString.pas unit

Purpose: This unit will patch the System.pas RTL to use a custom NON OLE COMPATIBLE WideString type, NOT using the slow Windows API, but FastMM4 (without COW)
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Functions or procedures implemented in the SynFastWideString unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>WideStringFree</td>
<td>This low-level helper can be used to free a WideString returned by COM/OLE</td>
<td>1349</td>
</tr>
</tbody>
</table>

procedure WideStringFree(var TrueBSTRWideStringVariable: WideString);

This low-level helper can be used to free a WideString returned by COM/OLE
- WideString instances created with this unit can be safely sent to any COM/OLE object, as soon as they are constant parameters, but not a "var" parameter or a callback function result
- any WideString instance returned by a COM object should NOT be released by Delphi automatically, since the following would create a memory error:
  TrueBSTRWideStringVariable := ''; if you are using SynFastWideString, you should use this procedure to release true BSTR
- if you are using SynFastWideString, you should use this procedure to release true BSTR WideString instance, as such:
  type
    _Catalog = interface(IDispatch)
      // this method will be safe to use with our unit
      function Create(const ConnectString: WideString): OleVariant; safecall;
      // this method won't be safe, since it returns a true BSTR as WideString
      function GetObjectOwner(const ObjectName: WideString; ObjectType: OleVariant; ObjectTypeId: OleVariant): WideString; safecall;
    end;
  ...;
  function CheckCatalogOwner(const catalog: _Catalog): string;
  var bstr: WideString;
  begin
    try // force manual handling of this true BSTR instance Lifetime
      bstr := catalog.GetObjectOwner('name',null,null);
      result := bstr; // conversion to string will work
    finally
      WideStringFree(bstr); // manual release, and set bstr := nil
    end;
  end;
  ...

- do a regular TrueBSTRWideStringVariable := '' since Delphi 2009, or call the low-level oleaut32.dll API for older versions, as expected by COM
27.21. SynGdiPlus.pas unit

**Purpose**: GDI+ library API access
- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic
- make available most useful GDI+ drawing methods
- allows Antialiased rending of any EMF file using GDI+
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The *SynGdiPlus* unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.3.2</td>
<td>A reporting feature, with full preview and export as PDF or TXT files, shall be integrated</td>
<td>2550</td>
</tr>
</tbody>
</table>

**SynGdiPlus class hierarchy**

**Objects implemented in the *SynGdiPlus* unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TGdipBitmapData</td>
<td>Data as retrieved by GdipFull.BitmapLockBits</td>
<td>1351</td>
</tr>
<tr>
<td>TGDIPlus</td>
<td>Handle picture related GDI+ library calls</td>
<td>1351</td>
</tr>
<tr>
<td>TGDIPlusFull</td>
<td>Handle most GDI+ library calls</td>
<td>1353</td>
</tr>
<tr>
<td>TGdipPointF</td>
<td>GDI+ floating point coordinates for a point</td>
<td>1351</td>
</tr>
<tr>
<td>TGdipRect</td>
<td>GDI+ integer coordinates rectangles</td>
<td>1351</td>
</tr>
<tr>
<td>TGdipRectF</td>
<td>GDI+ floating point coordinates rectangles</td>
<td>1351</td>
</tr>
<tr>
<td>TGifImage</td>
<td>Sub class to handle .GIF file extension</td>
<td>1353</td>
</tr>
<tr>
<td>TImageAttributes</td>
<td>An object wrapper to handle gdi+ image attributes</td>
<td>1351</td>
</tr>
</tbody>
</table>
**Synopsis mORMot Framework**  
**Software Architecture Design 1.18**  
**Date: September 16, 2020**

### Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TJpegImage</td>
<td>Sub class to handle .JPG file extension</td>
<td>1353</td>
</tr>
<tr>
<td>TPngImage</td>
<td>Sub class to handle .PNG file extension</td>
<td>1353</td>
</tr>
<tr>
<td>TSynLibrary</td>
<td>An object wrapper to load dynamically a library</td>
<td>1351</td>
</tr>
<tr>
<td>TSynPicture</td>
<td>GIF, PNG, TIFF and JPG pictures support using GDI+ library</td>
<td>1352</td>
</tr>
<tr>
<td>TTiffImage</td>
<td>Sub class to handle .TIF file extension</td>
<td>1353</td>
</tr>
</tbody>
</table>

**TGdipRect** = record

- **GDI+ integer coordinates rectangles**
- use width and height instead of right and bottom

**TGdipRectF** = record

- **GDI+ floating point coordinates rectangles**
- use width and height instead of right and bottom

**TGdipPointF** = record

- **GDI+ floating point coordinates for a point**

**TGdipBitmapData** = record

- **Data as retrieved by GdipFull.BitmapLockBits**

**TImageAttributes** = class(TObject)

- **An object wrapper to handle gdi+ image attributes**

**TSynLibrary** = class(TObject)

- **An object wrapper to load dynamically a library**

**function** Exists: boolean;

- **Return TRUE if the library and all procedures were found**

**TGDIPlus** = class(TSynLibrary)

- **Handle picture related GDI+ library calls**

  **Used for DI-2.3.2 (page 2550).**

  - **constructor** Create(const aDllFileName: TFileName); reintroduce;
    - **Load the GDI+ library and all needed procedures**
    - **returns TRUE on success**
    - library is loaded dynamically, therefore the executable is able to launch before Windows XP, but GDI+ functions (e.g. GIF, PNG, TIFF and JPG pictures support) won't be available in such case
  
  - **destructor** Destroy; override;
    - **Unload the GDI+ library**
function DrawAntiAliased(Source: TMetafile; ScaleX: integer=100; ScaleY: integer=100; aSmoothing: TSmoothingMode=smAntiAlias; aTextRendering: TTextRenderingHint=trhClearTypeGridFit): TBitmap; overload;

Draw the corresponding EMF metafile into a bitmap created by the method
- this default TGDIPlus implementation uses GDI drawing only
- use a TGDIPlusFull instance for true GDI+ AntiAliased drawing
- you can specify a zoom factor by the ScaleX and ScaleY parameters in percent: e.g. 100 means 100%, i.e. no scaling
- returned image is a DIB (device-independent bitmap)

Used for DI-2.3.2 (page 2550).

procedure DrawAntiAliased(Source: TMetafile; Dest: HDC; R: TRect; aSmoothing: TSmoothingMode=smAntiAlias; aTextRendering: TTextRenderingHint=trhClearTypeGridFit); overload; virtual;

Draw the corresponding EMF metafile into a given device context
- this default implementation uses GDI drawing only
- use TGDIPlusFull overridden method for true GDI+ AntiAliased drawing

Used for DI-2.3.2 (page 2550).

procedure RegisterPictures;

Registers the .jpg .jpeg .gif .png .tif .tiff file extensions to the program
- TPicture can now load such files
- you can just launch Gdip.RegisterPictures to initialize the GDI+ library

TSynPicture = class(TGraphic)

GIF, PNG, TIFF and JPG pictures support using GDI+ library
- cf @http://msdn.microsoft.com/en-us/library/ms536393 for available image formats

function GetImageFormat: TGDI PPPictureType;

Guess the picture type from its internal format
- return gptBMP if no format is found

class function IsPicture(const FileName: TFileName): TGraphicClass;

Return TRUE if the supplied filename is a picture handled by TSynPicture

function RectNotBiggerThan(MaxPixelsForBiggestSide: Integer): TRect;

Calculate a TRect which fit the specified maximum pixel number
- if any side of the picture is bigger than the specified pixel number, the TRect is sized down in order than the biggest size if this value

function SaveAs(Stream: TStream; Format: TGDI PPPictureType; CompressionQuality: integer=80; IfBitmapSetResolution: single=0): TGdipStatus;

Save the picture into any GIF/PNG/JPG/TIFF format
- CompressionQuality is used for gptJPG format saving and is expected to be from 0 to 100; for gptTIFF format, use ord(TGDI PEncoderValue) to define the parameter; by default, will use ord(evCompressionLZW) to save the TIFF picture with LZW - for gptTIFF, only valid values are ord(evCompressionLZW), ord(evCompressionCCITT3), ord(evCompressionCCITT4), ord(evCompressionRle) and ord(evCompressionNone)
function ToBitmap: TBitmap;
Create a bitmap from the corresponding picture
- kind of returned image is DIB (device-independent bitmap)

property NativeImage: THandle read fImage;
Return the GDI+ native image handle

TPngImage = class(TSynPicture)
Sub class to handle .PNG file extension

TJpegImage = class(TSynPicture)
Sub class to handle .JPG file extension

procedure SaveToStream(Stream: TStream); override;
Implements the saving feature

property CompressionQuality: integer read fCompressionQuality write fCompressionQuality;
The associated encoding quality (from 0 to 100)
- set to 80 by default

TGifImage = class(TSynPicture)
Sub class to handle .GIF file extension

TTiffImage = class(TSynPicture)
Sub class to handle .TIF file extension
- GDI+ seems not able to load all Tiff file formats, depending on the Windows version and third-party libraries installed
- this overridden class implements multiple pages

function GetPageCount:integer;
Retrieve the number of pages in the TIFF file

procedure ExtractPage(index: integer; wBMP: TBitmap);
Extract a page from the TIFF and assign it to a bitmap

property ActivePageIndex: integer read fActivePage write SelectPage;
Multi-page
- default Frame/Page Index is 0

TGDIPlusFull = class(TGDIPlus)
Handle most GDI+ library calls
- an instance of this object is initialized by this unit: you don't have to create a new instance
**constructor** Create(aDllFileName: TFileName='');

*Load the GDI+ library and all needed procedures*
- returns TRUE on success
- library is loaded dynamically, therefore the executable is able to launch before Windows XP, but GDI+ functions (e.g. GIF, PNG, TIFF and JPG pictures support or AntiAliased drawing) won't be available
- if no GdiPlus.dll file name is available, it will search the system for the most recent version of GDI+ (either GDIPPLUS.DLL in the current directory, either the Office 2003 version, either the OS version - 1.1 is available only since Vista and Seven; XP only shipped with version 1.1)

**function** ConvertToEmfPlus(Source: TMetafile; Dest: HDC; aSmoothing: TSmoothingMode=smAntiAlias; aTextRendering: TTextRenderingHint=trhClearTypeGridFit): THandle;

*Convert a supplied EMF metafile into a EMF+ (i.e. GDI+ metafile)*
- i.e. allows antialiased drawing of the EMF metafile
- if GDI+ is not available or conversion failed, return 0
- return a metafile handle, to be released after use (e.g. with DrawImageRect) by DisposeImage()

**function** MetaFileToStream(Source: TMetafile): IStream;

*Internal method used for GDI32 metafile loading*

**procedure** DrawAntiAliased(Source: TMetafile; Dest: HDC; R: TRect; aSmoothing: TSmoothingMode=smAntiAlias; aTextRendering: TTextRenderingHint=trhClearTypeGridFit); overload; override;

*Draw the corresponding EMF metafile into a given device context*
- this overridden implementation handles GDI+ AntiAliased drawing
- if GDI+ is not available, it will use default GDI32 function

**property** ForceInternalConvertToEmfPlus: boolean read fForceInternalConvertToEmfPlus write fForceInternalConvertToEmfPlus;

*Can be set to true if to force DrawAntiAliased() method NOT to use native GDI+ 1.1 conversion, even if available*
- we found out that GDI+ 1.1 was not as good as our internal conversion function written in Delphi, e.g. for underlined fonts or font fallback
- programs can set this property to true to avoid using GDI+ 1.1

**property** ForceUseDrawString: boolean read fUseDrawString write fUseDrawString;

*If TRUE, text will be rendered using DrawString and not DrawDriverString if the content has some chars non within 0000-05ff Unicode BMP range*
- less accurate for individual character positioning (e.g. justification), but will handle UniScribe positioning and associated font fallback
- is disable by default, and enabled only for chars >= $600
- note that the GdipConvertToEmfPlus() GDI+ 1.1 method does not handle font fallback, so our internal conversion is more accurate thanks to this parameter

**property** NativeConvertToEmfPlus: boolean read getNativeConvertToEmfPlus;

*Return true if DrawAntiAliased() method will use native GDI+ conversion, i.e. if GDI+ installed version is 1.1*

**Types implemented in the SynGdiPlus unit**

TEmfType = ( etEmf0, etEmf1, etEmf2, etEmfOnly, etEmfPlusOnly, etEmfPlusDual );

GDI+ types of conversion from EMF to EMF+
TFillMode = ( fmAlternate, fmWinding );

* GDI+ available filling modes *

TGDIPCombineMode = ( cmReplace, cmIntersect, cmUnion, cmXor, cmExclude, cmComplement );

* Region combine mode (used by SetClipRegion, etc.)*


* The optional TIFF compression levels*
- use e.g. ord(evCompressionCCITT4) to save a TIFF picture as CCITT4

TGDIPPictureType = ( gptGIF, gptPNG, gptJPG, gptBMP, gptTIF );

* Allowed types for image saving *

TgdipPointFArray = array[0..1000] of TGdipPointF;

* GDI+ floating point coordinates for an array of points *


* GDI+ error codes *

TLockModeOption = ( lmRead, lmWrite, lmUserInputBuf );

* GDI+ lock mode for GdipFull.BitmapLockBits *

TLockModeOptions = set of TLockModeOption;

* GDI+ lock mode settings for GdipFull.BitmapLockBits *

TSmoothingMode = ( smDefault, smHighSpeed, smHighQuality, smNone, smAntiAlias );

* GDI+ line drawing smoothing types *

TTTextRenderingHint = ( trhDefault, trhSingleBitPerPixelGridFit, trhSingleBitPerPixel, trhAntiAliasGridFit, trhAntiAlias, trhClearTypeGridFit );

* GDI+ text rendering smoothing types *

TUnit = ( uWorld, uDisplay, uPixel, uPoint, uInch, uDocument, uMillimeter, uGdi );

* GDI+ available coordinates units *

**Constants implemented in the SynGdiPlus unit**

GDIPPictureExt: array [TGDIPPictureType] of TFileName = ('.gif', '.png', '.jpg', '.bmp', '.tif');

* The corresponding file extension for every saving format type *

**Functions or procedures implemented in the SynGdiPlus unit**
### Functions or procedures

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitmapToRawByteString</td>
<td>Helper to save a specified TBitmap into GIF/PNG/JPG/TIFF format</td>
<td>1356</td>
</tr>
<tr>
<td>DrawEmfGdip</td>
<td>Draw the specified GDI TMetaFile (emf) using the GDI-plus antialiased engine</td>
<td>1356</td>
</tr>
<tr>
<td>ExpectGDIPlusFull</td>
<td>Will set global Gdip instance from a TGDIPlusFull, if available</td>
<td>1357</td>
</tr>
<tr>
<td>GdipLock</td>
<td>Enter global critical section for safe use of SynGdiPlus from multiple threads</td>
<td>1357</td>
</tr>
<tr>
<td>GdipTest</td>
<td>Test function</td>
<td>1357</td>
</tr>
<tr>
<td>GdipUnlock</td>
<td>Leave global critical section for safe use of SynGdiPlus from multiple threads</td>
<td>1357</td>
</tr>
<tr>
<td>JpegRecompress</td>
<td>Recompress a JPEG binary in-place</td>
<td>1357</td>
</tr>
<tr>
<td>LoadFrom</td>
<td>Helper function to create a bitmap from any GIF/PNG/JPG/TIFF/EMF/WMF file</td>
<td>1357</td>
</tr>
<tr>
<td>LoadFrom</td>
<td>Helper function to create a bitmap from any EMF content</td>
<td>1357</td>
</tr>
<tr>
<td>LoadFromRawByteString</td>
<td>Helper to load a specified graphic from GIF/PNG/JPG/TIFF format content</td>
<td>1357</td>
</tr>
<tr>
<td>PictureName</td>
<td>Retrieve a ready to be displayed name of the supplied Graphic Class</td>
<td>1357</td>
</tr>
<tr>
<td>SaveAs</td>
<td>Helper to save a specified graphic into GIF/PNG/JPG/TIFF format</td>
<td>1358</td>
</tr>
<tr>
<td>SaveAs</td>
<td>Helper to save a specified graphic into GIF/PNG/JPG/TIFF format</td>
<td>1358</td>
</tr>
<tr>
<td>SaveAsRawByteString</td>
<td>Helper to save a specified graphic into GIF/PNG/JPG/TIFF format</td>
<td>1358</td>
</tr>
</tbody>
</table>

**function** BitmapToRawByteString(Bitmap: TBitmap; **out** DataRawByteString; Format: TGDIPPictureType; CompressionQuality: integer=80; MaxPixelsForBiggestSide: cardinal=0; BitmapSetResolution: single=0): TGdipStatus;

*Helper to save a specified TBitmap into GIF/PNG/JPG/TIFF format*

- CompressionQuality is only used for gptJPG format saving and is expected to be from 0 to 100
- if MaxPixelsForBiggestSide is set to something else than 0, the resulting picture biggest side won’t exceed this pixel number
- this method is thread-safe (using GdipLock/GdipUnlock globals)

**procedure** DrawEmfGdip(aHDC: HDC; Source: TMetaFile; var R: TRect; ForceInternalAntiAliased: boolean; ForceInternalAntiAliasedFontFallBack: boolean=false);

*Draw the specified GDI TMetaFile (emf) using the GDI-plus antialiased engine*

- by default, no font fall-back is implemented (for characters not included within the font glyphs), but you may force it via the corresponding parameter (used to set the TGDIPlusFull.ForceUseDrawString property)
- this method is thread-safe (using GdipLock/GdipUnlock globals)
procedure ExpectGDIPlusFull(ForceInternalAntiAliased: boolean=true; ForceInternalAntiAliasedFontFallBack: boolean=true);

Will set global Gdip instance from a TGDIPPlusFull, if available
- this GDI+ 1.1 version (i.e. gdiplus11.dll) allows proper TMetaFile antialiasing, as requested by
LoadFrom and DrawEmfGdip functions

procedure GdipLock;

Enter global critical section for safe use of SynGdiPlus from multiple threads

procedure GdipTest(const JpegFile: TFileName);

Test function

procedure GdipUnlock;

Leave global critical section for safe use of SynGdiPlus from multiple threads

function JpegRecompress(const jpeg: AnsiString; quality: integer=80): AnsiString;

Recompress a JPEG binary in-place
- no sizing is done, but a bitmap is created from the supplied JPEG, and re-compressed as JPEG
  using the specified quality
- may be used to ensure a JPEG binary is a JPEG is a JPEG
- this method is thread-safe (using GdipLock/GdipUnlock globals)

function LoadFrom(const MetaFile: TMetaFile): TBitmap; overload;

Helper function to create a bitmap from any EMF content
- the file is drawn with a special antialiased GDI+ drawing method (if the global Gdip var is a
  TGDIPPlusFull instance)
- this method is thread-safe (using GdipLock/GdipUnlock globals)

function LoadFrom(const FileName: TFileName): TBitmap; overload;

Helper function to create a bitmap from any GIF/PNG/JPG/TIFF/EMF/WMF file
- if file extension is .EMF, the file is drawn with a special antialiased GDI+ drawing method (if the
  global Gdip var is a TGDIPPlusFull instance)
- this method is thread-safe (using GdipLock/GdipUnlock globals)

function LoadFromRawByteString(const Picture: AnsiString): TBitmap;

Helper to load a specified graphic from GIF/PNG/JPG/TIFF format content
- this method is thread-safe (using GdipLock/GdipUnlock globals)

function PictureName(Pic: TGraphicClass): string;

Retrieve a ready to be displayed name of the supplied Graphic Class

procedure SaveAs(Graphic: TPersistent; Stream: TStream; Format: TGDIPPictureType;
CompressionQuality: integer=80; MaxPixelsForBiggestSide: cardinal=0;
BitmapSetResolution: single=0); overload;

Helper to save a specified graphic into GIF/PNG/JPG/TIFF format
- CompressionQuality is only used for gptJPG format saving and is expected to be from 0 to 100
- if MaxPixelsForBiggestSide is set to something else than 0, the resulting picture biggest side
  won’t exceed this pixel number
- this method is thread-safe (using GdipLock/GdipUnlock globals)
procedure SaveAs(Graphic: TPersistent; const FileName: TFileName; Format: TGDIPPictureType; CompressionQuality: integer=80; MaxPixelsForBiggestSide: cardinal=0; BitmapSetResolution: single=0); overload;

   Helper to save a specified graphic into GIF/PNG/JPG/TIFF format
   - CompressionQuality is only used for gpJPG format saving and is expected to be from 0 to 100
   - if MaxPixelsForBiggestSide is set to something else than 0, the resulting picture biggest side
     won't exceed this pixel number
   - this method is thread-safe (using GdipLock/GdipUnlock globals)

procedure SaveAsRawByteString(Graphic: TPersistent; out DataRawByteString; Format: TGDIPPictureType; CompressionQuality: integer=80; MaxPixelsForBiggestSide: cardinal=0; BitmapSetResolution: single=0);

   Helper to save a specified graphic into GIF/PNG/JPG/TIFF format
   - CompressionQuality is only used for gpJPG format saving and is expected to be from 0 to 100
   - if MaxPixelsForBiggestSide is set to something else than 0, the resulting picture biggest side
     won't exceed this pixel number
   - this method is thread-safe (using GdipLock/GdipUnlock globals)

Variables implemented in the SynGdiPlus unit

Gdip: TGDIPPlus = nil;

   GDI+ library instance
   - only initialized at program startup if the NOTSYNPICTUREREGISTER is NOT defined (which is not
     the default)
   - Gdip.Exists return FALSE if the GDI+ library is not available in this operating system (e.g. on
     Windows 2000) nor the current executable folder
   - you can run ExpectGDIPPlusFull to ensure you use GDI+ 1.1

GdipCS: TRTLCriticalSection;

   Mutex used by GdipLock/GdipUnlock
27.22. SynLizard.pas unit

_Purpose_: Lizard (LZ5) compression routines (statically linked for FPC)
- licensed under a MPL/GPL/LGPL tri-license; original Lizard is BSD 2-Clause

Units used in the _SynLizard_ unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
</tbody>
</table>

![SynLizard class hierarchy](image)

Objects implemented in the _SynLizard_ unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSynLizard</td>
<td>Lizard (formerly LZ5) lossless compression algorithm</td>
<td>1359</td>
</tr>
<tr>
<td>TSynLizardDynamic</td>
<td>Try to load Lizard as an external library</td>
<td>1361</td>
</tr>
</tbody>
</table>

TSynLizard = _class_(TObject)

*Lizard (formerly LZ5) lossless compression algorithm*
- provides efficient compression with very fast decompression
- this class implements direct low-level access to the Lizard API - consider using AlgoLizard/AlgoLizardFast global instances for easier use
compress: function(src, dst: pointer; srcSize, maxDstSize, compressionLevel: integer): integer; cdecl;

Compresses srcSize bytes from src into already allocated dst buffer of size maxDstSize - which should be >= Lizard_compressBound(srcSize)
- returns number of bytes written into dst (necessarily <= maxDstSize), or 0 if compression fails
due to too small maxDstSize, <0 on other failure
- compressionLevel is from LIZARD_MIN_CLEVEL (10) to LIZARD_MAX_CLEVEL(49), any value <10 (e.g. 0) will use 17, and value >49 will use 49

<table>
<thead>
<tr>
<th>Lev</th>
<th>Comp</th>
<th>Decomp</th>
<th>CompSize</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7332 MB/s</td>
<td>8719 MB/s</td>
<td>211947520</td>
<td>100.00</td>
</tr>
<tr>
<td>10</td>
<td>346 MB/s</td>
<td>2610 MB/s</td>
<td>103402971</td>
<td>48.79</td>
</tr>
<tr>
<td>12</td>
<td>103 MB/s</td>
<td>2458 MB/s</td>
<td>86232422</td>
<td>40.69</td>
</tr>
<tr>
<td>15</td>
<td>50 MB/s</td>
<td>2552 MB/s</td>
<td>81187330</td>
<td>38.31</td>
</tr>
<tr>
<td>19</td>
<td>3.84 MB/s</td>
<td>2497 MB/s</td>
<td>77416400</td>
<td>36.53</td>
</tr>
<tr>
<td>21</td>
<td>1.17 MB/s</td>
<td>1795 MB/s</td>
<td>89239174</td>
<td>42.10</td>
</tr>
<tr>
<td>23</td>
<td>30 MB/s</td>
<td>1778 MB/s</td>
<td>81097176</td>
<td>38.26</td>
</tr>
<tr>
<td>26</td>
<td>6.63 MB/s</td>
<td>1734 MB/s</td>
<td>74503695</td>
<td>35.15</td>
</tr>
<tr>
<td>29</td>
<td>1.37 MB/s</td>
<td>1634 MB/s</td>
<td>68694227</td>
<td>32.41</td>
</tr>
<tr>
<td>30</td>
<td>246 MB/s</td>
<td>989 MB/s</td>
<td>85727429</td>
<td>48.45</td>
</tr>
<tr>
<td>32</td>
<td>94 MB/s</td>
<td>1244 MB/s</td>
<td>76929454</td>
<td>36.30</td>
</tr>
<tr>
<td>35</td>
<td>47 MB/s</td>
<td>1435 MB/s</td>
<td>73850400</td>
<td>34.84</td>
</tr>
<tr>
<td>39</td>
<td>2.94 MB/s</td>
<td>1592 MB/s</td>
<td>69887522</td>
<td>32.94</td>
</tr>
<tr>
<td>41</td>
<td>126 MB/s</td>
<td>961 MB/s</td>
<td>76100661</td>
<td>35.91</td>
</tr>
<tr>
<td>43</td>
<td>28 MB/s</td>
<td>1101 MB/s</td>
<td>79055653</td>
<td>33.48</td>
</tr>
<tr>
<td>46</td>
<td>6.25 MB/s</td>
<td>1073 MB/s</td>
<td>65413061</td>
<td>30.86</td>
</tr>
<tr>
<td>49</td>
<td>1.27 MB/s</td>
<td>1064 MB/s</td>
<td>68079215</td>
<td>28.63</td>
</tr>
</tbody>
</table>

compressBound: function(inputSize: integer): integer; cdecl;

Maximum size that Lizard compression may output in a "worst case" scenario

compress_extState: function(state: pointer; src, dst: pointer; srcSize, maxDstSize, compressionLevel: integer): integer; cdecl;

Compresses using an external pre-allocated state buffer

decompress_safe: function(src, dst: pointer; srcSize, maxDstSize: integer): integer; cdecl;

Decompresses srcSize bytes from src into already allocated dst buffer
- returns number of bytes written to dst (<= maxDstSize), or <=0 on failure
- this function is protected against buffer overflow exploits

decompress_safe_partial: function(src, dst: pointer; srcSize, targetDstSize, maxDstSize: integer): integer; cdecl;

Partial decompression srcSize bytes from src into already allocated dst buffer
- returns number of bytes written to dst (<= maxDstSize), or <=0 on failure
- number can be <targetDstSize should the compressed block to decode be smaller
- this function is protected against buffer overflow exploits

sizeofState: function(compressionLevel: integer): integer; cdecl;

How much memory must be allocated for compress_extState()

versionNumber: function: integer; cdecl;

Version number of the linked Lizard library

constructor Create; virtual;

/ will initialize the library
TSynLizardDynamic = class(TSynLizard)

Try to load Lizard as an external library
- static linking is currently available only on FPC Win32/64 and Linux32/64
- this class is expected to access Lizard1-32.dll/Lizard1-64.dll files for Delphi, e.g. as such:
  TSynLizardDynamic.AlgoRegister;

constructor Create(const aLibraryFile: TFileName = ''; aRaiseNoException: boolean = false); reintroduce;
  Will first search in the executable folder, then within the system path
  - raise an Exception if the library file is not found, or not valid - unless aRaiseNoException is set to true

destructor Destroy; override;
  Unload the external library

class function AlgoRegister: boolean;
  Ensure Lizard compression is available
  - returns TRUE if Lizard compression is available
  - if there is a local Lizard1-32.dll/Lizard1-64.dll file, try to load it

property LibraryName: TFileName read fLibraryName;
  The loaded library file name

property Loaded: boolean read fLoaded;
  Set to TRUE if Create successed
  - may be used if aRaiseNoException parameter has been defined

Constants implemented in the SynLizard unit

LIZARD_DEFAULT_CLEVEL = 0;
  Default compression level for TSynLizard.compress
  - 0 value will let the library use level 17 - slow but efficient - method
  - as used by AlgoLizard global TSynCompress instance

LIZARD_HUFFMAN_CLEVEL = 41;
  Fast huffman compression level for TSynLizard.compress
  - better compression ratio than LIZARD_DEFAULT_CLEVEL, better compression speed, but slower decompression

LIZARD_LIB_NAME = 'Lizard1-32.dll';
  Default TSynLizardDynamic file name
  - mainly for Delphi, since FPC will use static linked .o files under Windows and Linux Intel 32/64 bits
  - to be downloaded from from https://synopse.info/files/SynLizardLibs.7z

LIZARD_MAX_CLEVEL = 49;
  Maximum compression level for TSynLizard.compress

LIZARD_MIN_CLEVEL = 10;
  Minimum compression level for TSynLizard.compress
  - as used by AlgoLizardFast global TSynCompress instance
Variables implemented in the *SynLizard* unit

<table>
<thead>
<tr>
<th>Variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AlgoLizard</strong>: TAlgoCompress;</td>
<td><em>Implement Lizard compression in level 17 (LIZARD_DEFAULT_CLEVEL) as AlgoID=4</em></td>
</tr>
<tr>
<td></td>
<td>- is set by TSynLizard.Create, so available e.g. if library is statically linked, or once TSynLizardDynamic.Create has been successfully called</td>
</tr>
<tr>
<td><strong>AlgoLizardFast</strong>: TAlgoCompress;</td>
<td><em>Implement Lizard compression in level 10 (LIZARD_MIN_CLEVEL) as AlgoID=5</em></td>
</tr>
<tr>
<td></td>
<td>- is set by TSynLizard.Create, so available e.g. if library is statically linked, or once TSynLizardDynamic.Create has been successfully called</td>
</tr>
<tr>
<td><strong>AlgoLizardHuffman</strong>: TAlgoCompress;</td>
<td><em>Implement Lizard compression in level 41 (LIZARD_HUFFMAN_CLEVEL) as AlgoID=6</em></td>
</tr>
<tr>
<td></td>
<td>- is set by TSynLizard.Create, so available e.g. if library is statically linked, or once TSynLizardDynamic.Create has been successfully called</td>
</tr>
<tr>
<td><strong>Lizard</strong>: TSynLizard;</td>
<td><em>Direct access to the low-level Lizard (LZ5) library</em></td>
</tr>
<tr>
<td></td>
<td>- is defined by default if Lizard was statically linked (under FPC)</td>
</tr>
<tr>
<td></td>
<td>- otherwise, you should execute explicitly:</td>
</tr>
<tr>
<td></td>
<td>if Lizard = nil then</td>
</tr>
<tr>
<td></td>
<td>Lizard := TSynLizardDynamic.Create;</td>
</tr>
</tbody>
</table>
27.23. SynLog.pas unit

**Purpose**: Logging functions used by Synopse projects
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynLog unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>SynCommons</em></td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><em>SynLZ</em></td>
<td>SynLZ Compression routines</td>
<td>1393</td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><em>SynTable</em></td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

**SynLog class hierarchy**

Objects implemented in the SynLog unit
<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESynLogSilent</td>
<td>An exception which wouldn't be logged and intercepted by this unit</td>
<td>1367</td>
</tr>
<tr>
<td>ISynLog</td>
<td>A generic interface used for logging a method</td>
<td>1367</td>
</tr>
<tr>
<td>ISynLogCallback</td>
<td>A mORMot-compatible callback definition</td>
<td>1368</td>
</tr>
<tr>
<td>TAutoLockerDebug</td>
<td>Reference-counted block code critical section with context logging</td>
<td>1382</td>
</tr>
<tr>
<td>TFileStreamWithoutWriteError</td>
<td>File stream which ignores I/O write errors</td>
<td>1376</td>
</tr>
<tr>
<td>TSynLog</td>
<td>A per-family and/or per-thread log file content</td>
<td>1376</td>
</tr>
<tr>
<td>TSynLogCallback</td>
<td>Store a subscribe to ISynLogCallback</td>
<td>1368</td>
</tr>
<tr>
<td>TSynLogCallbacks</td>
<td>Can manage a list of ISynLogCallback registrations</td>
<td>1368</td>
</tr>
<tr>
<td>TSynLogExceptionInfo</td>
<td>Storage of the information associated with an intercepted exception</td>
<td>1387</td>
</tr>
<tr>
<td>TSynLogFamily</td>
<td>Regroup several logs under an unique family name</td>
<td>1370</td>
</tr>
<tr>
<td>TSynLogFile</td>
<td>Used to parse a .log file, as created by TSynLog, into high-level data</td>
<td>1383</td>
</tr>
<tr>
<td>TSynLogFileProc</td>
<td>Used by TSynLogFile to refer to a method profiling in a .log file</td>
<td>1382</td>
</tr>
<tr>
<td>TSynLogFileView</td>
<td>Used to parse a .log file and process into VCL/LCL/FMX</td>
<td>1386</td>
</tr>
<tr>
<td>TSynLogSettings</td>
<td>Store simple log-related settings</td>
<td>1369</td>
</tr>
<tr>
<td>TSynLogThreadContext</td>
<td>Thread-specific internal context used during logging</td>
<td>1376</td>
</tr>
<tr>
<td>TSynLogThreadRecursion</td>
<td>TSynLogThreadContext will define a dynamic array of such information</td>
<td>1375</td>
</tr>
<tr>
<td>TSynMapFile</td>
<td>Retrieve a .map file content, to be used e.g. with TSynLog to provide</td>
<td>1365</td>
</tr>
<tr>
<td></td>
<td>additional debugging information</td>
<td></td>
</tr>
<tr>
<td>TSynMapSymbol</td>
<td>A debugger symbol, as decoded by TSynMapFile from a .map file</td>
<td>1364</td>
</tr>
<tr>
<td>TSynMapUnit</td>
<td>A debugger unit, as decoded by TSynMapFile from a .map file</td>
<td>1365</td>
</tr>
</tbody>
</table>

**TSynMapSymbol** = `packed record`  
*A debugger symbol, as decoded by TSynMapFile from a .map file*

- **Name**: RawUTF8;  
  *Symbol internal name*

- **Start**: integer;  
  *Starting offset of this symbol in the executable*  
  - addresses are integer, since map be <0 in Kylix .map files

- **Stop**: integer;  
  *End offset of this symbol in the executable*  
  - addresses are integer, since map be <0 in Kylix .map files
**TSynMapUnit** = packed record

A debugger unit, as decoded by TSynMapFile from a .map file

- **Addr**: TIntegerDynArray;
  - Start code address of each source code line

- **FileName**: RawUTF8;
  - Associated source file name

- **Line**: TIntegerDynArray;
  - List of all mapped source code lines of this unit

- **Symbol**: TSynMapSymbol;
  - Name, Start and Stop of this Unit

**TSynMapFile** = class(TObject)

Retrieve a .map file content, to be used e.g. with TSynLog to provide additional debugging information

-original .map content can be saved as .mab file in a more optimized format

**constructor** Create(const aExeName: TFileName=''; MabCreate: boolean=true);

Get the available debugging information

- if aExeName is specified, will use it in its search for .map/.mab
- if aExeName is not specified, will use the currently running .exe/.dll
- it will first search for a .map matching the file name: if found, will be read to retrieve all necessary debugging information - a .mab file will be also created in the same directory (if MabCreate is TRUE)
- if .map is not not available, will search for the .mab file
- if no .mab is available, will search for a .mab appended to the .exe/.dll
- if nothing is available, will log as hexadecimal pointers, without debugging information

**function** AbsoluteToOffset(aAddressAbsolute: PtrUInt): integer;

Compute the relative memory address from its absolute (pointer) value

**class function** FindFileName(const unitname: RawUTF8): TFileName;

Returns the file name of

- if unitname = '', returns the main file name of the current executable

**function** FindLocation(aAddressAbsolute: PtrUInt): RawUTF8; overload;

Return the symbol location according to the supplied absolute address

- i.e. unit name, symbol name and line number (if any), as plain text
- returns '' if no match found

**class function** FindLocation(exc: ESynException): RawUTF8; overload;

Return the symbol location according to the supplied ESynException

- i.e. unit name, symbol name and line number (if any), as plain text
- under FPC, currently calls BacktraceStrFunc() which may be very slow

**class function** FindStackTrace(const Ctxt: TSynLogExceptionContext): TRawUTF8DynArray;

Return the low-level stack trace exception information into human-friendly text
function FindSymbol(aAddressOffset: integer): integer;

Retrieve a symbol according to a relative code address
- use fast O(log n) binary search

function FindUnit(aAddressOffset: integer; out LineNumber: integer): integer; overload;

Retrieve an unit and source line, according to a relative code address
- use fast O(log n) binary search

function FindUnit(const aUnitName: RawUTF8): integer; overload;

Retrieve an unit information, according to the unit name
- will search within Units array

class function FromCurrentExecutable: TSynMapFile;

Returns the global TSynMapFile instance associated with the current executable

function SaveToFile(const aFileName: TFileName=''): TFileName;

Save all debugging information in the .mab custom binary format
- if no file name is specified, it will be saved as ExeName.mab or DllName.mab
- this file content can be appended to the executable via SaveToExe method
- this function returns the created file name

class procedure Log(W: TTextWriter; aAddressAbsolute: PtrUInt; AllowNotCodeAddr: boolean);

Add some debugging information about the supplied absolute memory address
- will create a global TSynMapFile instance for the current process, if necessary
- if no debugging information is available (.map or .mab), will write the raw address pointer as hexadecimal
- under FPC, currently calls BacktraceStrFunc() which may be very slow

procedure SaveToExe(const aExeName: TFileName);

Append all debugging information to an executable (or library)
- the executable name must be specified, because it's impossible to write to the executable of a running process
- this method will work for .exe and for .dll (or .ocx)

procedure SaveToJson(const aJsonFile: TFileName; aHumanReadable: Boolean=false); overload;

Save all debugging information as a JSON file
- may be useful from debugging purposes

procedure SaveToJson(W: TTextWriter); overload;

Save all debugging information as JSON content
- may be useful from debugging purposes

procedure SaveToStream(aStream: TStream);

Save all debugging information in our custom binary format

property FileName: TFileName read fMapFile;

The associated file name

property HasDebugInfo: boolean read fHasDebugInfo;

Equals true if a .map or .mab debugging information has been loaded
property Symbols: TSynMapSymbolDynArray read fSymbol;
All symbols associated to the executable

property Units: TSynMapUnitDynArray read fUnit;
All units, including line numbers, associated to the executable

ESynLogSilent = class(ESynException)
An exception which wouldn't be logged and intercepted by this unit
- only this exact class will be recognized by TSynLog: inheriting it will trigger the interception, as
any other regular exception

ISynLog = interface(IUnknown)
A generic interface used for logging a method
- you should create one TSynLog instance at the beginning of a block code using TSynLog.Enter:
the ISynLog will be released automatically by the compiler at the end of the method block, marking
its execution end
- all logging expect UTF-8 encoded text, i.e. usually English text

function Instance: TSynLog;
Retrieve the associated logging instance

procedure Log(Level: TSynLogInfo; const aName: RawUTF8; aTypeInfo: pointer; const aValue; Instance: TObject); overload;
Call this method to add the content of most low-level types to the log at a specified level
- TSynLog will handle enumerations and dynamic array; TSQLLog will be able to write
TObject/TSQLRecord and sets content as JSON

procedure Log(Level: TSynLogInfo=sllTrace); overload;
Call this method to add the caller address to the log at the specified level
- if the debugging info is available from TSynMapFile, will log the unit name, associated symbol
and source code line

procedure Log(Level: TSynLogInfo; const TextFmt: RawUTF8; const TextArgs: array of const; Instance: TObject=nil); overload;
Call this method to add some information to the log at a specified level
- will use TTextWriter.Add(...,twOnSameLine) to append its content
- % = #37 indicates a string, integer, floating-point, class parameter to be appended as text (e.g.
class name), any variant as JSON...
- note that cardinal values should be type-casted to Int64() (otherwise the integer mapped value
will be transmitted, therefore wrongly)
- if Instance is set, it will log the corresponding class name and address (to be used if you didn't
call TSynLog.Enter() method first)

procedure Log(Level: TSynLogInfo; const Text: RawUTF8; Instance: TObject=nil;
TextTruncateAtLength: integer=maxInt); overload;
Call this method to add some information to the log at a specified level
- if Instance is set and Text is not ", it will log the corresponding class name and address (to be
used e.g. if you didn't call TSynLog.Enter() method first)
- if Instance is set and Text is ", will behave the same as Log(Level,Instance), i.e. write the
Instance as JSON content
procedure Log(Level: TSynLogInfo; Instance: TObject); overload;
   Call this method to add the content of an object to the log at a specified level
   - TSynLog will write the class and hexa address - TSQLog will write the object JSON content

procedure LogLines(Level: TSynLogInfo; LinesToLog: PUTF8Char; aInstance: TObject=nil; const IgnoreWhenStartWith: PAnsiChar=nil);
   Call this method to add some multi-line information to the log at a specified level
   - LinesToLog content will be added, one line per one line, delimited by #13#10 (CRLF)
   - if a line starts with IgnoreWhenStartWith (already uppercase), it won’t be added to the log content (to be used e.g. with ‘--’ for SQL statements)

ISynLogCallback = interface(IInvokable)
   A mORMot-compatible callback definition
   - used to notify a remote mORMot server via interface-based services for any incoming event, using e.g. TSynLogCallbacks.Subscribe

procedure Log(Level: TSynLogInfo; const Text: RawUTF8);
   Each line of the TTextWriter internal instance will trigger this method
   - the format is similar to TOnTextWriterEcho, as defined in SynCommons
   - an initial call with Level=sllNone and the whole previous Text may be transmitted, if ReceiveExistingKB is set for TSynLogCallbacks.Subscribe()

TSynLogCallback = record
   Store a subscribe to ISynLogCallback

TSynLogCallbacks = class(TSynPersistentLock)
   Can manage a list of ISynLogCallback registrations

   Registration: TSynLogCallbackDynArray;
      Direct access to the registration storage

   Registrations: TDynArray;
      High-level access to the registration storage

   TrackedLog: TSynLogFamily;
      The TSynLog family actually associated with those callbacks

   constructor Create(aTrackedLog: TSynLogFamily); reintroduce;
      Initialize the registration storage for a given TSynLogFamily instance

   destructor Destroy; override;
      Finalize the registration storage for a given TSynLogFamily instance

   function OnEcho(Sender: TTextWriter; Level: TSynLogInfo; const Text: RawUTF8): boolean;
      Notify a given log event
      - matches the TOnTextWriterEcho signature
function Subscribe(const Levels: TSynLogInfos; const Callback: ISynLogCallback; ReceiveExistingKB: cardinal=0): integer; virtual;

Register a callback for a given set of log levels
- you can specify a number of KB of existing log content to send to the monitoring tool, before the actual real-time process

procedure Unsubscribe(const Callback: ISynLogCallback); virtual;

Unregister a callback previously registered by Subscribe()

property Count: integer read fCount;

How many registrations are currently defined

TSynLogSettings = class(TSynPersistent)

Store simple log-related settings
- see also TDDDLogSettings in dddInfraSettings.pas and TSynDaemonSettings in mORMotService.pas, which may be more integrated

constructor Create; override;

Set some default values

procedure SetLog(aLogClass: TSynLogClass = nil);

Define the log information into the supplied TSynLog class
- if you don't call this method, the logging won't be initiated

property DestinationPath: TFileName read fDestinationPath write fDestinationPath;

Allow to customize where the logs should be written
- default is the system log folder (e.g. /var/log on Linux)

property Levels: TSynLogInfos read fLevels write fLevels;

The log levels to be used for the log file
- i.e. a combination of none or several logging event
- if "+*" is serialized, unneeded sllNone won't be part of the set
- default is LOG_STACKTRACE

property LogClass: TSynLogClass read fLogClass;

Read-only access to the TSynLog class, if SetLog() has been called

property RotateFileCount: integer read fRotateFileCount write fRotateFileCount;

How many files will be rotated (default is 2)
**TSynLogFamily = class(TObject)**

*Regroup several logs under an unique family name*
- you should usually use one family per application or per architectural module: e.g. a server application may want to log in separate files the low-level Communication, the DB access, and the high-level process
- initialize the family settings before using them, like in this code:
  ```pascal
  with TSynLogDB.Family do begin
    Level := LOG_VERBOSE;
    PerThreadLog := ptOneFilePerThread;
    DestinationPath := 'C:\Logs';
  end;
  ```
- then use the logging system inside a method:
  ```pascal
  procedure TMyDB.MyMethod;
  var ILog: ISynLog;
  begin
    ILog := TSynLogDB.Enter(self, 'MyMethod');
    // do some stuff
    ILog.Log(sllInfo, 'method called');
  end;
  ```

**constructor Create(aSynLog: TSynLogClass);**

*Initialize for a TSynLog class family*
- add it in the global SynLogFileFamily[] list

**destructor Destroy; override;**

*Release associated memory*
- will archive older DestinationPath/*.log files, according to ArchiveAfterDays value and ArchivePath

**function GetExistingLog(MaximumKB: cardinal): RawUTF8;**

*Can be used to retrieve up to a specified amount of KB of existing log*
- expects a single file to be opened for this family
- will retrieve the log content for the current file, truncating the text up to the specified number of KB (an up to 128 MB at most)

**function SynLog: TSynLog;**

*Retrieve the corresponding log file of this thread and family*
- creates the TSynLog if not already existing for this current thread

**procedure EchoRemoteStart(aClient: TObject; const aClientEvent: TOnTextWriterEcho; aClientOwnedByFamily: boolean);**

*Register one object and one echo callback for remote logging*
- aClient is typically a mORMot's TSQLHttpClient or a TSynLogCallbacks instance as defined in this unit
- if aClientOwnedByFamily is TRUE, its life time will be manage by this TSynLogFamily: it will stay alive until this TSynLogFamily is destroyed, or the EchoRemoteStop() method called
- aClientEvent should be able to send the log row to the remote server

**procedure EchoRemoteStop;**

*Stop echo remote logging*
- will free the aClient instance supplied to EchoRemoteStart

---

**SynLog.pas unit - Rev. 1.18**

Page 1371 of 2552
procedure OnThreadEnded(Sender: TThread);

Callback to notify the current logger that its thread is finished
- method follows TNotifyThreadEvent signature, which can be assigned to
  TSynBackgroundThreadAbstract.OnAfterExecute
- is called e.g. by TSQLRest.EndCurrentThread

property ArchiveAfterDays: Integer read fArchiveAfterDays write fArchiveAfterDays;
Number of days before OnArchive event will be called to compress or delete deprecated files
- will be set by default to 7 days
- will be used by Destroy to call OnArchive event handler on time

property ArchivePath: TFileName read fArchivePath write fArchivePath;
The folder where old log files must be compressed
- by default, is in the executable folder, i.e. the same as DestinationPath
- the 'log' sub folder name will always be appended to this value
- will then be used by OnArchive event handler to produce, with the current file date year and
  month, the final path (e.g. 'ArchivePath\Log\YYYYMM*.log.synlz' or
  'ArchivePath\Log\YYYYMM.zip')

property AutoFlushTimeOut: cardinal read fAutoFlush write fAutoFlush;
The time (in seconds) after which the log content must be written on disk, whatever the current
content size is
- by default, the log file will be written for every 4 KB of log (see BufferSize property) - this will
  ensure that the main application won't be slow down by logging
- in order not to loose any log, a background thread can be created and will be responsible of
  flushing all pending log content every period of time (e.g. every 10 seconds)

property BufferSize: integer read fBufferSize write fBufferSize;
The internal in-memory buffer size, in bytes
- this is the number of bytes kept in memory before flushing to the hard drive; you can call
  TSynLog.Flush method or set AutoFlushTimeOut to true in order to force the writing to disk
- is set to 4096 by default (4 KB is the standard hard drive cluster size)

property CustomFileName: TFileName read fCustomFileName write fCustomFileName;
Can be used to customized the default file name
- by default, the log file name is computed from the executable name (and the computer name if
  IncludeComputerNameInFileName is true)
- you can specify your own file name here, to be used instead
- this file name should not contain any folder, nor file extension (which are set by
  DestinationPath and DefaultExtension properties)

property DefaultExtension: TFileName read fDefaultExtension write fDefaultExtension;
The file extension to be used
- is '.log' by default

property DestinationPath: TFileName read fDestinationPath write SetDestinationPath;
The folder where the log must be stored
- by default, is in the executable folder
property EchoCustom: TOnTextWriterEcho read fEchoCustom write SetEchoCustom;

*Can be set to a callback which will be called for each log line*
- could be used with a third-party logging system
- EchoToConsole or EchoCustom can be activated separately
- you may even disable the integrated file output, via NoFile := true

property EchoToConsole: TSynLogInfos read fEchoToConsole write SetEchoToConsole;

*If the some kind of events shall be echoed to the console*
- note that it will slow down the logging process a lot (console output is slow by nature under Windows, but may be convenient for interactive debugging of services, for instance
- this property shall be set before any actual logging, otherwise it will have no effect
- can be set e.g. to LOG_VERBOSE in order to echo every kind of events
- EchoCustom or EchoToConsole can be activated separately

property EchoToConsoleUseJournal: boolean read fEchoToConsoleUseJournal write SetEchoToConsoleUseJournal;

*For Linux with journald*
- if true: redirect all EchoToConsole logging into journald service
- such logs can be exported into a format which can be viewed by our LogView tool using a command (replacing UNIT with your unit name and PROCESS with the executable name):
  "journalctl -u UNIT --no-hostname -o short-iso-precise --since today | grep "PROCESS\[.*\]": . > todaysLog.log"

property EndOfLineCRLF: boolean read fEndOfLineCRLF write fEndOfLineCRLF;

*Define how the logger will emit its line feed*
- by default (FALSE), a single LF (#10) char will be written, to save storage space
- you can set this property to TRUE, so that CR+LF (#13#10) chars will be appended instead
- TSynLogFile class and our LogView tool will handle both patterns

property ExceptionIgnore: TList read fExceptionIgnore;

*You can add some exceptions to be ignored to this list*
- for instance, EConvertError may be added to the list, as such:
  TSQLLog.Family.ExceptionIgnore.Add(EConvertError);
- you may also trigger ESynLogSilent exceptions for silent process
- see also ExceptionIgnoreCurrentThread property, if you want a per-thread filtering of all exceptions

property ExceptionIgnoreCurrentThread: boolean read

*GetExceptionIgnoreCurrentThread write SetExceptionIgnoreCurrentThread;

-Allow to (temporarily) ignore exceptions in the current thread*
- this property will affect all TSynLogFamily instances, for the current thread
- may be used in a try...finally block e.g. when notifying the exception to a third-party service, or during a particular process
- see also ExceptionIgnore property - which is also checked in addition to this flag

property FileExistsAction: TSynLogExistsAction read fFileExistsAction write fFileExistsAction;

*How existing log file shall be handled*
**property** HighResolutionTimestamp: boolean read fHRTimestamp write fHRTimestamp;
- If TRUE, will log high-resolution time stamp instead of ISO 8601 date and time
- this is less human readable, but allows performance profiling of your application on the customer side (using TSynLog.Enter methods)
- set to FALSE by default, or if RotateFileCount and RotateFileSizeKB / RotateFileDailyAtHour are set (the high resolution frequency is set in the log file header, so expects a single file)

**property** Ident: integer read fIdent;
- Index in global SynLogFileFamily[] and SynLogFileIndexThreadVar[] lists

**property** IncludeComputerNameInFileName: boolean read fIncludeComputerNameInFileName write fIncludeComputerNameInFileName;
- If TRUE, the log file name will contain the Computer name - as '(MyComputer)'

**property** Level: TSynLogInfos read fLevel write SetLevel;
- The current level of logging information for this family
- can be set e.g. to LOG_VERBOSE in order to log every kind of events

**property** LevelStackTrace: TSynLogInfos read fLevelStackTrace write fLevelStackTrace;
- The levels which will include a stack trace of the caller
- by default, contains sllStackTrace, sllException, sllExceptionOS plus
- sllError, sllFail, sllLastError, sllDDDError for Delphi only - since FPC BacktraceStrFunc() function is very slow
- exceptions will always trace the stack

**property** LocalTimestamp: boolean read fLocalTimestamp write fLocalTimestamp;
- By default, time logging will use error-safe UTC values as reference
- you may set this property to TRUE to store local time instead

**property** NoEnvironmentVariable: boolean read fNoEnvironmentVariable write fNoEnvironmentVariable;
- Force no environment variables to be written to the log file
- may be usefull if they contain some sensitive information

**property** NoFile: boolean read fNoFile write fNoFile;
- Force no log to be written to any file
- may be usefull in conjunction e.g. with EchoToConsole or any other third-party logging component

**property** OnArchive: TSynLogArchiveEvent read fOnArchive write fOnArchive;
- Event called to archive the .log content after a defined delay
- Destroy will parse DestinationPath folder for *.log files matching ArchiveAfterDays property value
- you can set this property to EventArchiveDelete in order to delete deprecated files, or EventArchiveSynLZ to compress the .log file into our propertary SynLZ format: resulting file name will be ArchivePath\log\YYYYMM\*.log.synlz (use FileUnSynLZ function to uncompress it)
- if you use SynZip.EventArchiveZip, the log files will be archived in ArchivePath\log\YYYYMM.zip
- the aDestinationPath parameter will contain 'ArchivePath\log\YYYYMM\'
- this event handler will be called one time per .log file to archive, then one last time with aOldLogFileDate="" in order to close any pending archive (used e.g. by EventArchiveZip to open the .zip only once)
property OnBeforeException: TSynLogOnBeforeException read fOnBeforeException write fOnBeforeException;

You can let exceptions be ignored from a callback
- if set and returns true, the given exception won't be logged
- execution of this event handler is protected via the logs global lock
- may be handy e.g. when working with code triggering a lot of exceptions (e.g. Indy), where ExceptionIgnore could be refined

property OnRotate: TSynLogRotateEvent read fOnRotate write fOnRotate;

Event called to perform a custom file rotation
- will be checked by TSynLog.PerformRotation to customize the rotation process and do not perform the default step, if the callback returns TRUE

property PerThreadLog: TSynLogPerThreadMode read fPerThreadLog write fPerThreadLog;

Define how thread will be identified during logging process
- by default, ptMergedInOneFile will indicate that all threads are logged in the same file, in occurrence order (so multi-thread process on server side may be difficult to interpret)
- if RotateFileCount and RotateFileSizeKB/RotateFileDailyAtHour are set, will be ignored (internal thread list shall be defined for one process)

property RotateFileCount: cardinal read fRotateFileCount write fRotateFileCount;

Auto-rotation of logging files
- set to 0 by default, meaning no rotation
- can be set to a number of rotating files: rotation and compression will happen, and main file size will be up to RotateFileSizeKB number of bytes, or when RotateFileDailyAtHour time is reached
- if set to 1, no .synlz backup will be created, so the main log file will be restarted from scratch when it reaches RotateFileSizeKB size or when RotateFileDailyAtHour time is reached
- if set to a number > 1, some rotated files will be compressed using the SynLZ algorithm, and will be named e.g. as MainLogFileName.0.synlz .. MainLogFileName.7.synlz for RotateFileCount=9 (total count = 9, including 1 main log file and 8 .synlz files)

property RotateFileDailyAtHour: integer read fRotateFileAtHour write fRotateFileDailyAtHour;

Fixed hour of the day where logging files rotation should be performed
- by default, equals -1, meaning no rotation
- you can set a time value between 0 and 23 to force the rotation at this specified hour
- is not used if RotateFileCount is left to its default 0

property RotateFileNoCompression: boolean read fRotateFileNoCompression write fRotateFileNoCompression;

If set to TRUE, no .synlz will be created at rotation but plain #.log file

property RotateFileSizeKB: cardinal read fRotateFileSize write fRotateFileSize;

Maximum size of auto-rotated logging files, in kilo-bytes (per 1024 bytes)
- specify the maximum file size upon which .synlz rotation takes place
- is not used if RotateFileCount is left to its default 0
property StackTraceLevel: byte read fStackTraceLevel write fStackTraceLevel;
    The recursive depth of stack trace symbol to write
    - used only if exceptions are handled, or by sllStackTraceLevel
    - default value is 30, maximum is 255
    - if stOnlyAPI is defined as StackTraceUse under Windows XP, maximum value may be around
      60, due to RtlCaptureStackBackTrace() API limitations

property StackTraceUse: TSynLogStackTraceUse read fStackTraceUse write fStackTraceUse;
    How the stack trace shall use only the Windows API
    - the class will use low-level RtlCaptureStackBackTrace() API to retrieve the call stack: in some
      cases, it is not able to retrieve it, therefore a manual walk of the stack can be processed - since
      this manual call can trigger some unexpected access violations or return wrong positions, you
      can disable this optional manual walk by setting it to stOnlyAPI
    - default is stManualAndAPI, i.e. use RtlCaptureStackBackTrace() API and perform a manual stack
      walk if the API returned no address (or <3); but within the IDE, it will use stOnlyAPI, to ensure no
      annoying AV occurs

property SynLogClass: TSynLogClass read fSynLogClass;
    The associated TSynLog class

property SynLogClassName: string read GetSynLogClassName;
    The associated TSynLog class

property WithInstancePointer: boolean read fWithInstancePointer write fWithInstancePointer;
    If TRUE, will log the pointer with an object instance class if available
    - set to TRUE by default, for better debugging experience

property WithUnitName: boolean read fWithUnitName write fWithUnitName;
    If TRUE, will log the unit name with an object instance if available
    - unit name is available from RTTI if the class has published properties
    - set to TRUE by default, for better debugging experience

TSynLogThreadRecursion = record
    TSynLogThreadContext will define a dynamic array of such information
        - used by TSynLog.Enter methods to handle recursivity calls tracing
    Caller: PTrUInt;
        The caller address, ready to display stack trace dump if needed
    EnterTimestamp: Int64;
        The timestamp at enter time
    Instance: TObject;
        Associated class instance to be displayed
    MethodName: PUTF8Char;
        Method name (or message) to be displayed
        - may be a RawUTF8 if MethodNameLocal=mnEnterOwnMethodName
MethodNameLocal: (mnAlways, mnEnter, mnLeave, mnEnterOwnMethodName);

   If the method name is local, i.e. shall not be displayed at Leave()

RefCount: integer;

   Internal reference count used at this recursion level by TSynLog._AddRef

TSynLogThreadContext = record

   Thread-specific internal context used during logging
   - this structure is a hashed-per-thread variable

   ID: TThreadID;
               The corresponding Thread ID

   Recursion: array of TSynLogThreadRecursion;
               Used by TSynLog.Enter methods to handle recursivity calls tracing

   RecursionCapacity: integer;
               Number of items available in Recursion[]
               - faster than length(Recursion)

   RecursionCount: integer;
               Number of items stored in Recursion[]

   ThreadName: RawUTF8;
               The associated thread name

TFileStreamWithoutWriteError = class(TFileStream)

   File stream which ignores I/O write errors
   - in case disk space is exhausted, TFileStreamWithoutWriteError.WriteBuffer won't throw any exception, so application will continue to work
   - used by TSynLog to let the application continue with no exception, even in case of a disk/partition full of logs

   function Write(const Buffer; Count: Longint): Longint; override;
               This overridden function returns Count, as if it was always successfull

TSynLog = class(TObject)

   A per-family and/or per-thread log file content
   - you should create a sub class per kind of log file

   TSynLogDB = class(TSynLog);

   - the TSynLog instance won't be allocated in heap, but will share a per-thread (if Family.PerThreadLog=ptOneFilePerThread) or global private log file instance
   - was very optimized for speed, if no logging is written, and even during log write (using an internal TTextWriter)
   - can use available debugging information via the TSynMapFile class, for stack trace logging for exceptions, sllStackTrace, and Enter/Leave labelling
<table>
<thead>
<tr>
<th>constructor Create(aFamily: TSynLogFamily=nil); virtual;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize for a TSynLog class instance</td>
</tr>
<tr>
<td>- WARNING: not to be called directly! Use Enter or Add class function instead</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>destructor Destroy; override;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release all memory and internal handles</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>class function Add: TSynLog;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve the current instance of this TSynLog class</td>
</tr>
<tr>
<td>- to be used for direct logging, without any Enter/Leave:</td>
</tr>
<tr>
<td>TSynLogDB.Add.Log(llError,'The % statement didn't work',[SQL]);</td>
</tr>
<tr>
<td>- to be used for direct logging, without any Enter/Leave (one parameter version - just the same as previous):</td>
</tr>
<tr>
<td>TSynLogDB.Add.Log(llError,'The % statement didn't work',SQL);</td>
</tr>
<tr>
<td>- is just a wrapper around Family.SynLog - the same code will work:</td>
</tr>
<tr>
<td>TSynLogDB.Family.SynLog.Log(llError,'The % statement didn't work',[SQL]);</td>
</tr>
</tbody>
</table>
**class function** Enter(aInstance: TObject=nil; aMethodName: PUTF8Char=nil; aMethodNameLocal: boolean=false): ISynLog; overload;

Handle generic method enter / auto-leave tracing

- this is the main method to be called within a procedure/function to trace:

```pascal
procedure TMyDB.SQLExecute(const SQL: RawUTF8);
var ILog: ISynLog;
begin
  ILog := TSynLogDB.Enter(self, 'SQLExecute');
  // do some stuff
  ILog.Log(sllInfo, 'SQL=%', [SQL]);
end;
```

- returning a ISynLog interface will allow you to have an automated sllLeave log created when the method is left (thanks to the hidden try..finally block generated by the compiler to protect the ISynLog var)
- it is convenient to define a local variable to store the returned ISynLog and use it for any specific logging within the method execution
- if you just need to access the log inside the method block, you may not need any ISynLog interface variable:

```pascal
procedure TMyDB.SQLFlush;
begin
  TSynLogDB.Enter(self, 'SQLFlush');
  // do some stuff
end;
```

- if no Method name is supplied, it will use the caller address, and will write it as hexa and with full unit and symbol name, if the debugging information is available (i.e. if TSynMapFile retrieved the .map content):

```pascal
procedure TMyDB.SQLFlush;
begin
  TSynLogDB.Enter(self);
  // do some stuff
end;
```

- note that supplying a method name is faster than using the .map content: if you want accurate profiling, it's better to use a method name or not to use a .map file - note that this method name shall be a constant, and not a locally computed variable, since it may trigger some random GPF at runtime - if it is a local variable, you can set aMethodNameLocal=true
- if TSynLogFamily.HighResolutionTimestamp is TRUE, high-resolution time stamp will be written instead of ISO 8601 date and time: this will allow performance profiling of the application on the customer side

- Enter() will write the class name (and the unit name for classes with published properties, if TSynLogFamily.WithUnitName is true) for both enter (+) and leave (-) events:

```pascal
20110325 19325801 + MyDBUnit.TMyDB(004E11F4).SQLExecute
20110325 19325801 info SQL=SELECT * FROM Table;
20110325 19325801 + 01.512.320
```

- note that due to a limitation (feature?) of the FPC compiler, you need to hold the returned value into a local ISynLog variable, as such:

```pascal
procedure TMyDB.SQLFlush;
var Log: ISynLog;
begin
  Log := TSynLogDB.Enter(self);
  // do some stuff
end; // here Log will be released
```

otherwise, the ISynLog instance would be released just after the Enter() call, so the timing won't match the method execution
class function Enter(const TextFmt: RawUTF8; const TextArgs: array of const; aInstance: TObject=nil): ISynLog; overload;

Handle method enter / auto-leave tracing, with some custom text
- this overloaded method would not write the method name, but the supplied text content,
  after expanding the parameters like FormatUTF8()
- it will append the corresponding sILLeave log entry when the method ends

class function Family: TSynLogFamily; overload;

Retrieve the family of this TSynLog class type

function LogClass: TSynLogClass;

The associated TSynLog class

class function Void: TSynLogClass;

Returns a logging class which will never log anything
- i.e. a TSynLog sub-class with Family.Level := []

procedure CloseLogFile;

Flush all log content to file and close the file

class procedure DebuggerNotify(Level: TSynLogInfo; const Format: RawUTF8; const Args: array of const);

Low-level method helper which can be called to make debugging easier
- log some warning message to the TSynLog family
- will force a manual breakpoint if tests are run from the IDE

procedure DisableRemoteLog(value: boolean);

Allow to temporary disable remote logging
- to be used within a try ... finally section:
  log.DisableRemoteLog(true);
  try
    log.Log(....); // won’t be propagated to the remote Log
  finally
    log.DisableRemoteLog(false);
  end;

procedure Flush(ForceDiskWrite: boolean);

Flush all log content to file
- if ForceDiskWrite is TRUE, will wait until written on disk (slow)

procedure ForceRotation;

Force log rotation; Can be used for example inside SIGHUP signal handler
procedure Log(Level: TSynLogInfo; aInstance: TObject); overload;

*Call this method to add the content of an object to the log at a specified level*
- this default implementation will just write the class name and its hexa pointer value, and handle TList, TCollections and TStrings - for instance:
  TSynLog.Add.Log(sllDebug,GarbageCollector);

will append this line to the log:
```
20110330 10010005 debug
{"TObjectList(00B1AD60)"}:["TObjectList(00B1AE20)","TObjectList(00B1AE80)"]
```
- if aInstance is an Exception, it will handle its class name and Message:
```
20110330 10010005 debug "EClassName(00C2129A)" :"Exception message"
```
- use TSQLLog from mORMot.pas unit to add the record content, written as human readable JSON

procedure Log(Level: TSynLogInfo; const aName: RawUTF8; aTypeInfo: pointer; const aValue; Instance: TObject); overload;

*Call this method to add the content of most low-level types to the log at a specified level*
- this overridden implementation will write the value content, written as human readable JSON:
  - handle dynamic arrays and enumerations
  - TSQLLog from mORMot.pas unit will be able to write TObject/TSQLRecord and sets content as JSON

procedure Log(Level: TSynLogInfo); overload;

*Call this method to add the caller address to the log at the specified level*
- if the debugging info is available from TSynMapFile, will log the unit name, associated symbol and source code line

procedure Log(Level: TSynLogInfo; const TextFmt: RawUTF8; const TextArg: RawUTF8; aInstance: TObject=nil); overload;

*Same as Log(Level,TextFmt,[]) but with one RawUTF8 parameter*

procedure Log(Level: TSynLogInfo; const TextFmt: RawUTF8; const TextArgs: array of const; aInstance: TObject=nil); overload;

*Call this method to add some information to the log at the specified level*
- will use TTextWriter.Add(...,twOnSameLine) to append its content
- % = #37 indicates a string, integer, floating-point, class parameter to be appended as text (e.g. class name), any variant as JSON...
- note that cardinal values should be type-casted to Int64() (otherwise the integer mapped value will be transmitted, therefore wrongly)

procedure Log(Level: TSynLogInfo; const Text: RawUTF8; aInstance: TObject=nil; TextTruncateAtLength: integer=maxInt); overload;

*Call this method to add some information to the log at the specified level*
- if Instance is set and Text is not ", it will log the corresponding class name and address (to be used e.g. if you didn't call TSynLog.Enter() method first) - for instance
  TSQLLog.Add.Log(sllDebug,'GarbageCollector',GarbageCollector);

will append this line to the log:
```
0000000000002DB9 debug TObjectList(00425E68) GarbageCollector
```
- if Instance is set and Text is ", will behave the same as Log(Level,Instance), i.e. write the Instance as JSON content
procedure Log(Level: TSynLogInfo; const TextFmt: RawUTF8; const TextArg: Int64; aInstance: TObject=nil); overload;

Same as Log(Level,TextFmt,[]) but with one Int64 parameter

procedure LogLines(Level: TSynLogInfo; LinesToLog: PUTF8Char; aInstance: TObject=nil; const IgnoreWhenStartWith: PAnsiChar=nil);

Call this method to add some multi-line information to the log at a specified level
- LinesToLog content will be added, one line per one line, delimited by 13#10 (CRLF)
- if a line starts with IgnoreWhenStartWith (already uppercase), it won't be added to the log content (to be used e.g. with '--' for SQL statements)

procedure LogThreadName(const Name: RawUTF8; IgnoreIfAlreadySet: boolean=false);

Allows to identify the current thread with a textual representation
- would append an sllInfo entry with "SetThreadName ThreadID=Name" text
- entry would also be replicated at the begining of any rotated log file

procedure NotifyThreadEnded;

You may call this method when a thread is ended
- should be called in the thread context which is about to terminate, in a situation where no other logging may occur from this thread any more
- it will release all thread-specific resource used by this TSynLog
- is called e.g. by TSQLRest.EndCurrentThread, via TSynLogFamily.OnThreadEnded

procedure Release;

Flush all log content to file, close the file, and release the instance
- you should never call the Free method directly, since the instance is registered in a global TObjectList and an access violation may occur at application closing; you can use this Release method if you are sure that you won't need this TSynLog instance any more
- ensure there is no pending Leave element in a stack-allocated ISynLog (see below)
- can be used e.g. to release the instance when finishing a thread when Family.PerThreadLog=ptOneFilePerThread:

```pascal
var
  TThreadLogger : TSynLogClass = TSynLog;

procedure TMyThread.Execute;
var log : ISynLog;
begin
  log := TThreadLogger.Enter(self);
  ...
  log := nil; // to force logging end of method
end;
```

property FileName: TFileName read fFileName;

The associated file name containing the log
- this is accurate only with the default implementation of the class: any child may override it with a custom logging mechanism

property FileSize: Int64 read GetFileSize;

The current size, in bytes, of the associated file containing the log

property GenericFamily: TSynLogFamily read fFamily;

The associated logging family
property ThreadContextCount: integer read fThreadContextCount;

The current number of thread contexts associated with this instance
- doesn't match necessary the number of threads of the process, but the threads which are still
  marked as active for this TSynLog
- a huge number may therefore not indicate a potential "out of memory" error, but a broken
  logic with missing NotifyThreadEnded calls

property Writer: TTextWriterWithEcho read fWriter;

Direct access to the low-level writing content
- should usually not be used directly, unless you ensure it is safe

TAutoLockerDebug = class(TAutoLocker)

Reference-counted block code critical section with context logging
- race conditions are difficult to track: you could use this TAutoLockerDebug instead of plain
  TAutoLocker class, to log some information at each Enter/Leave process, and track unexpected
  blocking issues.
- see also the global USELOCKERDEBUG conditional, defined in Synopse.inc, which may be used to
  enable verbose logging at compile time:
  fSafe: IAutoLocker;
  ...
  {$ifdef USELOCKERDEBUG}
  fSafe := TAutoLockerDebug.Create(fLogClass,aModel.Root); // more verbose
  {$else}
  fSafe := TAutoLocker.Create;
  {$endif}

constructor Create(aLog: TSynLogClass; const aIdentifier: RawUTF8); reintroduce;

Initialize the mutex, which would log its Enter/Leave process
- the supplied identifier should be a short text, able to specify the lock execution context, e.g.
  the resource which is actually protected
- an associated TSQLog instance should be specified as logging target

procedure Enter; override;

Enter the mutex

procedure Leave; override;

Leave the mutex

TSynLogFileProc = record

Used by TSynLogFile to refer to a method profiling in a .log file
- i.e. map a sllEnter/sllLeave event in the .log file

Index: cardinal;

The index of the sllEnter event in the TSynLogFile.fLevels[] array

ProperTime: cardinal;

The time elapsed in this method and not in nested methods
- computed from Time property, minus the nested calls
Time: cardinal;

*The associated time elapsed in this method (in micro seconds)*
- computed from the sllLeave time difference (high resolution timer)

**TSynLogFile = class(TMemoryMapText)**

*Used to parse a .log file, as created by TSynLog, into high-level data*
- this particular TMemoryMapText class will retrieve only valid event lines (i.e. will fill EventLevel[] for each line <> sllNone)
- Count is not the global text line numbers, but the number of valid events within the file (LinePointers/Line/Strings will contain only event lines) - it will not be a concern, since the .log header is parsed explicitly

**constructor** Create; **override;**

*Initialize internal structure*

**function** EventCount(const aSet: TSynLogInfos): integer;

*Return the number of matching events in the log*

**function** EventDateTime(aIndex: integer): TDateTime;

*Retrieve the date and time of an event*
- returns 0 in case of an invalid supplied index

**function** EventString(index: integer; const replaceTabs: RawUTF8=''; maxutf8len: Integer=0; includeFirstColumns: boolean=false): string;

*Retrieve the description text of an event, as native VCL string*
- returns "" if supplied index is out of range
- if the text is not truly UTF-8 encoded, would use the current system codepage to create a valid string
- you may specify a text to replace all #9 characters occurrences
- is used e.g. in TMainLogView.ListDrawCell

**function** LineContains(const aUpperSearch: RawUTF8; aIndex: Integer): Boolean; **override;**

*Returns TRUE if the supplied text is contained in the corresponding line*

**function** ThreadName(ThreadID, CurrentLogIndex: integer): RawUTF8;

*Returns the name of a given thread, according to the position in the log*

**function** ThreadNames(CurrentLogIndex: integer): TRawUTF8DynArray;

*Returns the name of all threads, according to the position in the log*
- result[0] stores the name of ThreadID = 1

**function** ThreadRows(ThreadID: integer): cardinal;

*Returns the number of occurrences of a given thread*

**procedure** AddInMemoryLine(const aNewLine: RawUTF8); **override;**

*Add a new line to the already parsed content*
- overridden method which would identify the freq=%,%,% pseudo-header

**procedure** GetDays(out Days: TDateTimeDynArray);

*Returns all days of this log file*
- only available for low-resolution timestamp, i.e. Freq=0
procedure LogProcSort(Order: TLogProcSortOrder);
    Sort the LogProc[] array according to the supplied order

property ComputerHost: RawUTF8 read fHost;
    The computer host name in which the process was running on

property CPU: RawUTF8 read fCPU;
    The computer CPU in which the process was running on
    - returns e.g. '1*0-15-1027'

property DayChangeIndex: TIntegerDynArray read fDayChangeIndex;
    The row indexes where the day changed
    - only available for low-resolution timestamp, i.e. Freq=0
    - if set, contains at least [0] if the whole log is over a single day

property DayCount: TIntegerDynArray read fDayCount;
    The number of rows for each DayChangeIndex[] value

property DetailedOS: RawUTF8 read fOSDetailed;
    The computer Operating System in which the process was running on
    - returns e.g. '2.3=5.1.2600' for Windows XP
    - under Linux, it will return the full system version, e.g. 'Ubuntu=Linux-3.13.0-43-generic#72-Ubuntu-SMP-Mon-Dec-8-19:35:44-UTC-2014'

property EventLevel: TSynLogInfoDynArray read fLevels;
    Retrieve the level of an event
    - is calculated by Create() constructor
    - EventLevel[] array index is from 0 to Count-1

property EventLevelUsed: TSynLogInfos read fLevelUsed;
    Retrieve all used event levels
    - is calculated by Create() constructor

property EventText[index: integer]: RawUTF8 read GetEventText;
    Retrieve the description text of an event
    - returns '' if supplied index is out of range
    - see also EventString() function, for direct VCL use

property EventThread: TWordDynArray read fThreads;
    Retrieve all event thread IDs
    - contains something if TSynLogFamily.PerThreadLog was ptIdentifiedInOnFile
    - for ptMergedInOnFile (default) or ptOneFilePerThread logging process, the array will be void (EventThread=nil)

property ExecutableDate: TDateTime read fExeDate;
    The associated executable build date and time

property ExecutableName: RawUTF8 read fExeName;
    The associated executable name (with path)
    - returns e.g. 'C:\Dev\lib\SQLite3\exe\TestSQL3.exe'

property ExecutableVersion: RawUTF8 read fExeVersion;
    The associated executable version
    - returns e.g. '0.0.0.0'
property Framework: RawUTF8 read fFramework;
    The associated framework information
    - returns e.g. 'TSQLLog 1.18.2765 ERTL FTS3'

property Freq: Int64 read fFreq;
    High-resolution time stamp frequency, as retrieved from log file header
    - equals 0 if date time resolution, >0 if high-resolution time stamp

property Headers: RawUTF8 read fHeaders;
    Custom headers, to be searched as .ini content

property InstanceName: RawUTF8 read fInstanceName;
    For a library, the associated instance name (with path)
    - returns e.g. 'C:\Dev\lib\SQLite3\exe\TestLibrary.dll'
    - for an executable, will be left void

property IntelCPU: TIntelCpuFeatures read fIntelCPU;
    The available CPU features, as recognized at program startup
    - is extracted from the last part of the CPU property text
    - you could use the overloaded ToText() function to show it in an human-friendly way

property LevelUsed: TSynLogInfos read fLevelUsed;
    All used event levels, as retrieved at log file content parsing

    Profiled methods information
    - is calculated by Create() constructor
    - will contain the sllEnter index, with the associated elapsed time
    - number of items in the array is retrieved by the LogProcCount property

property LogProcCount: integer read fLogProcCurrentCount;
    Number of profiled methods in this .log file
    - i.e. number of items in the LogProc[] array

    If the method information must be merged for the same method name

property LogProcOrder: TLogProcSortOrder read fLogProcSortInternalOrder;
    The current sort order

property OS: TWindowsVersion read fOS;
    The computer Operating System in which the process was running on
    - equals wUnknown on Linux or BSD - use DetailedOS instead

property RunningUser: RawUTF8 read fUser;
    The computer user name who launched the process

property ServicePack: integer read fOSServicePack;
    The Operating System Service Pack number
    - not defined on Linux or BSD - use DetailedOS instead

property StartDateTime: TDateTime read fStartDateTime;
    The date and time at which the log file was started
property ThreadsCount: cardinal read fThreadMax;
  The number of threads

property Wow64: boolean read fWow64;
  If the 32 bit process was running under WOW 64 virtual emulation

TSynLogFileView = class(TSynLogFile)
  Used to parse a .log file and process into VCL/LCL/FMX
  - would handle e.g. selection and search feature

function GetCell(aCol, aRow: integer; out aLevel: TSynLogInfo): string;
  Returns the ready-to-be text of a cell of the main TDrawGrid

function GetLineForClipboard(aRow: integer): string;
  Returns the ready-to-be copied text of a selected row

function GetLineForMemo(aRow, aTop, aBottom: integer): string;
  Returns the ready-to-be displayed text of one or several selected rows

function SearchEnterLeave(aRow: integer): integer;
  Search for the matching Enter/Leave item, from the current row index
  - returns -1 if no match was found

function SearchNextEvent(aEvent: TSynLogInfo; aRow: integer): integer;
  Search for the next matching TSynLogInfo, from the current row index
  - returns -1 if no match was found

function SearchNextSameThread(aRow: integer): integer;
  Search for the next matching thread, from the current row index
  - returns -1 if no match was found

function SearchNextSelected(aIndex: integer): integer;
  Search for the next row index, appearing after the supplied item index
  - returns -1 if no match was found

function SearchNextText(const aPattern: RawUTF8; aRow, aDelta: integer): integer;
  Search for the next matching text, from the current row index
  - returns -1 if no match was found

function SearchNextThread(aRow: integer): integer;
  Search for the next diverse thread, from the current row index
  - returns -1 if no match was found

function SearchPreviousSameThread(aRow: integer): integer;
  Search for the previous matching thread, from the current row index
  - returns -1 if no match was found

function SearchPreviousText(const aPattern: RawUTF8; aRow: integer): integer;
  Search for the previous matching text, from the current row index
  - returns -1 if no match was found
function SearchThread(aThreadID: word; aRow: integer): integer;
  Search for the next specified thread, from the current row index
  - returns -1 if no match was found

function Select(aRow: integer): integer; virtual;
  Fill all rows matching Events and Threads[] properties in Selected[]
  - you may specify the current selected row index, which would return the closest one after
    the selection has been applied

procedure AddInMemoryLine(const aNewLine: RawUTF8); override;
  Add a new line to the already parsed content
  - overridden method would add the inserted index to Selected[]

procedure SetAllThreads(enabled: boolean);
  Set all Threads[] to a specified value

property Events: TSynLogInfos read fEvents write fEvents;
  Define the current selection range, according to event kinds
  - once you have set Events and Threads[], call Select() to fill Selected[]

property Selected: TIntegerDynArray read fSelected;
  The row indexes of the selected entries

property SelectedCount: integer read fSelectedCount;
  How many entries are currently stored in Selected[]

property Threads[thread: integer]: boolean read GetThreads write SetThreads;
  Define the current selection range, according to a thread ID
  - here the supplied thread ID starts at 1
  - once you have set Events and Threads[], call Select() to fill Selected[]

TSynLogExceptionInfo = record
  Storage of the information associated with an intercepted exception
  - as returned by GetLastException() function
  Addr: RawUTF8;
    Ready-to-be-displayed text of the exception address
  Context: TSynLogExceptionContext;
    Low-level calling context
    - as used by TSynLogExceptionToStr callbacks
  Message: string;
    Associated Exception.Message content (if any)

Types implemented in the SynLog unit

PSynLogThreadContext = ^TSynLogThreadContext;
  Pointer to thread-specific context information

TLogProcSortOrder = ( soNone, soByName, soByOccurrence, soByTime, soByProperTime );
  Used by TSynLogFile.LogProcSort method

TSynLogArchiveEvent = function(const aOldLogFileName, aDestinationPath: TFileName):
boolean;

This event can be set for a TSynLogFamily to archive any deprecated log into a custom compressed format
- will be called by TSynLogFamily when TSynLogFamily.Destroy identify some outdated files
- the aOldLogFileFileName will contain the .log file with full path
- the aDestinationPath parameter will contain 'ArchivePath\log\YYYYMM'
- should return true on success, false on error
- example of matching event handler are EventArchiveDelete/EventArchiveSynLZ or EventArchiveZip in SynZip.pas
- this event handler will be called one time per .log file to archive, then one last time with aOldLogFileFileName="" in order to close any pending archive (used e.g. by EventArchiveZip to open the .zip only once)

TSynLogCallbackDynArray = array of TSynLogCallback;

Store the all subscribed ISynLogCallback

TSynLogClass = class of TSynLog;

Class-reference type (metaclass) of a TSynLog family
- since TSynLog classes store their information per type, you usually will store a reference to a logging family (i.e. logging settings) using a TSynLogClass variable, whereas TSynLog would point to the active logging instance

TSynLogExceptionInfoDynArray = array of TSynLogExceptionInfo;

Storage of the information associated with one or several exceptions
- as returned by GetLastExceptions() function

TSynLogExistsAction = ( acOverwrite, acAppend );

How file existing shall be handled during logging

TSynLogFileProcDynArray = array of TSynLogFileProc;

Used by TSynLogFile to refer to global method profiling in a .log file
- i.e. map all sllEnter/sllLeave event in the .log file

TSynLogFilter = ( lfNone, lfAll, lfErrors, lfExceptions, lfProfile, lfDatabase, lfClientServer, lfDebug, lfCustom, lfDDD );

A list of log events families, used to gather events by type

TSynLogOnBeforeException = function(const aExceptionContext: TSynLogExceptionContext; const aThreadName: RawUTF8): boolean of object;

Callback signature used by TSynLogFamily.OnBeforeException
- should return false to log the exception, or true to ignore it

TSynLogPerThreadMode = ( ptMergedInOneFile, ptOneFilePerThread, ptIdentifiedInOnFile, ptNoThreadProcess );

How threading is handled by the TSynLogFamily
- proper threading expects the TSynLog.NotifyThreadEnded method to be called when a thread is about to terminate, e.g. from TSQLRest.EndCurrentThread
- by default, ptMergedInOneFile will indicate that all threads are logged in the same file, in occurrence order
- if set to ptOneFilePerThread, it will create one .log file per thread
- if set to ptIdentifiedInOnFile, a new column will be added for each log row, with the corresponding ThreadID - LogView tool will be able to display per-thread logging, if needed - note that your application shall use a thread pool (just like all mORMot servers classes do), otherwise some random
hash collision may occur if Thread IDs are not recycled enough
- if set to ptNoThreadProcess, no thread information is gathered, and all Enter/Leave would be
merged into a single call - but it may be mandatory to use this option if TSynLog.NotifyThreadEnded
is not called (e.g. from legacy code), and that your process experiment instability issues

**TSynLogRotateEvent = function**(aLog: TSynLog; const aOldLogFileName: TFileName): boolean;

*This event can be set for a TSynLogFamily to customize the file rotation*
- will be called by TSynLog.PerformRotation
- should return TRUE if the function did process the file name
- should return FALSE if the function did not do anything, so that the caller should perform the
rotation as usual

**TSynLogStackTraceUse** = ( stManualAndAPI, stOnlyAPI, stOnlyManual );

*How stack trace shall be computed during logging*

**TSynMapSymbolDynArray** = array of TSynMapSymbol;

*A dynamic array of symbols, as decoded by TSynMapFile from a .map file*

**TSynMapUnitDynArray** = array of TSynMapUnit;

*A dynamic array of units, as decoded by TSynMapFile from a .map file*

**TSyslogFacility** = ( sfKern, sfUser, sfMail, sfDaemon, sfAuth, sfSyslog, sfLpr, sfNews,
sfUucp, sfClock, sfAuthpriv, sfFtp, sfNtp, sfAudit, sfAlert, sfCron, sfLocal0, sfLocal1,
sfLocal2, sfLocal3, sfLocal4, sfLocal5, sfLocal6, sfLocal7 );

*Syslog message facilities as defined by RFC 3164*

**TSyslogSeverity** = ( ssEmerg, ssAlert, ssCrit, ssErr, ssWarn, ssNotice, ssInfo, ssDebug);

*Syslog message severities as defined by RFC 5424*

**Constants implemented in the SynLog unit**

**LOG_CONSOLE_COLORS** = array[TsynLogInfo] of TConsoleColor = ( ccLightGray, ccWhite, ccLightGray, ccLightBlue, ccBrown, ccLightRed, ccGreen, ccGreen, ccLightRed, ccLightRed, ccLightGray, ccCyan, ccLightRed, ccBrown, ccBlue, ccLightCyan, ccMagenta, ccCyan, ccLightCyan, ccLightCyan, ccLightMagenta, ccMagenta, ccLightGray, ccLightGray, ccLightGray, ccLightMagenta, ccLightRed, ccWhite, ccLightBlue);

*Console colors corresponding to each logging level*
- SynCommons' TextColor()

**LOG_DEBUGERROR** = array[boolean] of TSynLogInfo = ( sllDebug, sllError );

*May be used to log as Debug or Error event, depending on an Error: boolean*

**LOG_FILTER** = array[TsynLogFilter] of TSynLogInfos = ( [], [sllNone].high(TSynLogInfo)],
[sllError, sllLastError, sllException, sllExceptionOS], [sllException, sllExceptionOS],
[sllEnter, sllLeave], [sllSQL, sllCache, sllDB], [sllClient, sllServer, sllServiceCall,
sllServiceReturn], [sllDebug, sllTrace, sllEnter],
[sllCustom1...sllCustom4], [sllDDDError, sllDDDInfo]);

*sllNone, sllInfo, sllDebug, sllTrace, sllWarning, sllError, sllEnter, sllLeave sllLastError, sllException, sllExceptionOS, sllMemory, sllStackTrace, sllFail, sllSQL, sllCache, sllResult, sllDB, sllHTTP, sllClient, sllServer, sllServiceCall, sllServiceReturn, sllUserAuth, sllCustom1, sllCustom2, sllCustom3, sllCustom4, sllNewRun, sllDDDError, sllDDDInfo, sllMonitoring how TLogFilter map TSynLogInfo events*
**TSynLog** unit - Rev. 1.18

### Functions or procedures implemented in the SynLog unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventArchiveDelete</td>
<td>A TSynLogArchiveEvent handler which will delete older .log files</td>
<td>1391</td>
</tr>
</tbody>
</table>

### LOG_INFOWARNING: array[boolean] of TSynLogInfo = (sllInfo, sllWarning);

*May be used to log as Info or Warning event, depending on an Error: boolean*

### LOG_LEVEL_COLORS: array[Boolean,TSynLogInfo] of integer = (...

- **RGB colors corresponding to each logging level**
  - matches the TColor values, as used by the VCL


- **The text equivalency of each logging level, as written in the log file**
  - PCardinal(@LOG_LEVEL_TEXT[L][3]) will be used for fast level matching so text must be unique for characters [3..6] - e.g. 'UST4'

### LOG_MAGIC = $ABA51051;

- The "magic" number used to identify .log.synlz compressed files, as created by TSynLogFamily.EventArchiveSynLZ

### LOG_STACKTRACE: TSynLogInfos = [sllException,sllExceptionOS, sllLastError,sllError,sllDDDError];

- **Contains the logging levels for which stack trace should be dumped**
  - which are mainly exceptions or application errors

### LOG_SYSLOG: array[TSynLogInfo] of TSyslogSeverity = (...

- **Used to convert a TSynLog event level into a syslog message severity**

### LOG_TRACEWARNING: array[boolean] of TSynLogInfo = (sllTrace, sllWarning);

*May be used to log as Trace or Warning event, depending on an Error: boolean*

### LOG_VERBOSE: TSynLogInfos = [succ(sllNone)..high(TSynLogInfo)];

*Can be set to TSynLogFamily.Level in order to log all available events*

### MAX_SYNLOGFAMILY = 15;

*Up to 16 TSynLogFamily, i.e. TSynLog children classes can be defined*
## Synopsis::ORMot Framework

**Software Architecture Design 1.18**  
**Date:** September 16, 2020

### Functions or procedures

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventArchiveSynLZ</td>
<td>A TSynLogArchiveEvent handler which will compress older .log files using our proprietary SynLZ format</td>
<td>1391</td>
</tr>
<tr>
<td>GetLastException</td>
<td>Makes a thread-safe copy of the latest intercepted exception</td>
<td>1391</td>
</tr>
<tr>
<td>GetLastExceptions</td>
<td>Returns a TDocVariant array of the latest intercepted exception texts</td>
<td>1392</td>
</tr>
<tr>
<td>GetLastExceptions</td>
<td>Makes a thread-safe copy of the latest intercepted exceptions</td>
<td>1392</td>
</tr>
<tr>
<td>GetLastExceptionText</td>
<td>Returns some text about the latest intercepted exception</td>
<td>1392</td>
</tr>
<tr>
<td>IsActiveLogFile</td>
<td>Check if the supplied file name is a currently working log file</td>
<td>1392</td>
</tr>
<tr>
<td>SyslogMessage</td>
<td>Append some information to a syslog message memory buffer</td>
<td>1392</td>
</tr>
<tr>
<td>ToCaption</td>
<td>Returns the ready-to-be displayed text of a TSynLogFilter value</td>
<td>1392</td>
</tr>
<tr>
<td>ToCaption</td>
<td>Returns the ready-to-be displayed text of a TSynLogInfo value</td>
<td>1392</td>
</tr>
<tr>
<td>ToText</td>
<td>Convert low-level exception information into some human-friendly text</td>
<td>1392</td>
</tr>
<tr>
<td>ToText</td>
<td>Returns the trimmed text value of a logging levels set</td>
<td>1392</td>
</tr>
<tr>
<td>ToText</td>
<td>SllNone, sllInfo, sllDebug, sllTrace, sllWarning, sllError, sllEnter, sllLeave, sllLastError, sllException, sllExceptionOS, sllMemory, sllStackTrace, sllFail, sllSQL, sllCache, sllResult, sllDB, sllHTTP, sllClient, sllServer, sllServiceCall, sllServiceReturn, sllUserAuth, sllCustom1, sllCustom2, sllCustom3, sllCustom4, sllNewRun, sllDDDError, sllDDDInfo, sllMonitoring); returns the trimmed text value of a logging level</td>
<td>1392</td>
</tr>
<tr>
<td>ToText</td>
<td>Returns a method event as text, using the .map/.mab information if available</td>
<td>1392</td>
</tr>
</tbody>
</table>

### Function Definitions

```pascal
function EventArchiveDelete(const aOldLogFileName, aDestinationPath: TFileName): boolean;
// A TSynLogArchiveEvent handler which will delete older .log files
```

```pascal
function EventArchiveSynLZ(const aOldLogFileName, aDestinationPath: TFileName): boolean;
// A TSynLogArchiveEvent handler which will compress older .log files using our proprietary SynLZ format
// - resulting file will have the .synlz extension and will be located in the aDestinationPath directory, i.e. TSynLogFamily.ArchivePath+'\log\YYYYMM\'
// - use UnSynLZ.dpr tool to uncompress it into .log textual file
// - SynLZ is much faster than zip for compression content, but proprietary
```

```pascal
function GetLastException(out info: TSynLogExceptionInfo): boolean;
// Makes a thread-safe copy of the latest intercepted exception
```

```pascal
procedure GetLastExceptions(out result: TSynLogExceptionInfoDynArray; Depth: integer=0); overload;
// Makes a thread-safe copy of the latest intercepted exceptions
```
function GetLastExceptions(Depth: integer=0): variant; overload;
  Returns a TDocVariant array of the latest intercepted exception texts
  - runs ToText() over all information returned by overloaded GetLastExceptions

function GetLastExceptionText: RawUTF8;
  Returns some text about the latest intercepted exception

function IsActiveLogFile(const aFileName: TFileName): boolean;
  Check if the supplied file name is a currently working log file
  - may be used to avoid e.g. infinite recursion when monitoring the log file

function SyslogMessage(facility: TSyslogFacility; severity: TSyslogSeverity; const msg, procid, msgid: RawUTF8; destbuffer: PUTF8Char; destsize: PtrInt; trimmsgfromlog: boolean): PtrInt;
  Append some information to a syslog message memory buffer
  - following https://tools.ietf.org/html/rfc5424 specifications
  - ready to be sent via UDP to a syslog remote server
  - returns the number of bytes written to destbuffer (which should have destsize > 127)

function ToCaption(event: TSynLogInfo): string; overload;
  Returns the ready-to-be displayed text of a TSynLogInfo value

function ToCaption(filter: TSynLogFilter): string; overload;
  Returns the ready-to-be displayed text of a TSynLogFilter value

function ToText(event: TSynLogInfo): RawUTF8; overload;
  sllNone, sllInfo, sllDebug, sllTrace, sllWarning, sllError, sllEnter, sllLeave, sllLastError, sllException, sllExceptionOS, sllMemory, sllStackTrace, sllFail, sllSQL, sllCache, sllResult, sllDB, sllHTTP, sllClient, sllServer, sllServiceCall, sllServiceReturn, sllUserAuth, sllCustom1, sllCustom2, sllCustom3, sllCustom4, sllNewRun, sllDDDError, sllDDDInfo, sllMonitoring); returns the trimmed text value of a logging level
  - i.e. 'Warning' for sllWarning

function ToText(const Event: TMethod): RawUTF8; overload;
  Returns a method event as text, using the .map/.mab information if available

function ToText(var info: TSynLogExceptionInfo): RawUTF8; overload;
  Convert low-level exception information into some human-friendly text

function ToText(events: TSynLogInfos): ShortString; overload;
  Returns the trimmed text value of a logging levels set

Variables implemented in the SynLog unit

GlobalCurrentHandleExceptionSynLog: TSynLog;
  Low-level variable used internally by this unit
  - do not access this variable in your code: defined here to allow inlining

GlobalThreadLock: TRTLCriticalSection;
  Low-level variable used internally by this unit
  - do not access this variable in your code: defined here to allow inlining

TSynLogTestLog: TSynLogClass = TSynLog;
  The kind of .log file generated by TSynTestsLogged
27.24. SynLZ.pas unit

*Purpose:* SynLZ Compression routines
- licensed under a MPL/GPL/LGPL tri-license; version 1.18

### Functions or procedures implemented in the *SynLZ* unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynLZcompress1</td>
<td>Optimized x86/x64 asm version of the 1st compression algorithm</td>
<td>1393</td>
</tr>
<tr>
<td>SynLZcompress1pas</td>
<td>1st compression algorithm uses hashing with a 32bits control word</td>
<td>1393</td>
</tr>
<tr>
<td>SynLZcompress2</td>
<td>2nd compression algorithm optimizing pattern copy</td>
<td>1393</td>
</tr>
<tr>
<td>SynLZcompressdestlen</td>
<td>Get uncompressed size from lz-compressed buffer (to reserve memory, e.g.)</td>
<td>1394</td>
</tr>
<tr>
<td>SynLZdecompress1</td>
<td>Optimized x86/x64 asm version of the 1st compression algorithm</td>
<td>1393</td>
</tr>
<tr>
<td>SynLZdecompress1partial</td>
<td>1st compression algorithm uses hashing with a 32bits control word</td>
<td>1393</td>
</tr>
<tr>
<td>SynLZdecompress1pas</td>
<td>1st compression algorithm uses hashing with a 32bits control word</td>
<td>1394</td>
</tr>
<tr>
<td>SynLZdecompress2</td>
<td>2nd compression algorithm optimizing pattern copy</td>
<td>1394</td>
</tr>
</tbody>
</table>

**function** SynLZcompress1(src: PAnsiChar; size: integer; dst: PAnsiChar): integer;
*Optimized x86/x64 asm version of the 1st compression algorithm*

**function** SynLZcompress1pas(src: PAnsiChar; size: integer; dst: PAnsiChar): integer;
*1st compression algorithm uses hashing with a 32bits control word*

**function** SynLZcompress2(src: PAnsiChar; size: integer; dst: PAnsiChar): integer;
*2nd compression algorithm optimizing pattern copy*
- this algorithm is a bit smaller, but slower, so the 1st method is preferred

**function** SynLZcompressdestlen(in_len: integer): integer;
*Get maximum possible (worse) compressed size for out_p*

**function** SynLZdecompress1(src: PAnsiChar; size: integer; dst: PAnsiChar): integer;
*Optimized x86/x64 asm version of the 1st compression algorithm*

**function** SynLZdecompress1partial(src: PAnsiChar; size: integer; dst: PAnsiChar; maxDst: integer): integer;
*1st compression algorithm uses hashing with a 32bits control word*
- this overload function is slower, but will allow to uncompress only the start of the content (e.g. to read some metadata header)
- it will also check for dst buffer overflow, so will be more secure than other functions, which expect the content to be verified (e.g. via CRC)
function SynLZdecompress1pas(src: PAnsiChar; size: integer; dst: PAnsiChar): integer;

1st compression algorithm uses hashing with a 32bits control word
- this is the fastest pure pascal implementation

function SynLZdecompress2(src: PAnsiChar; size: integer; dst: PAnsiChar): integer;

2nd compression algorithm optimizing pattern copy
- this algorithm is a bit smaller, but slower, so the 1st method is preferred

function SynLZdecompressdestlen(in_p: PAnsiChar): integer;

Get uncompressed size from lz-compressed buffer (to reserve memory, e.g.)
27.25. SynLZO.pas unit

_Purpose_: Fast LZO Compression routines
- licensed under a MPL/GPL/LGPL tri-license; version 1.13

### Functions or procedures implemented in the **SynLZO** unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CompressSynLZO</code></td>
<td>(de)compress a data content using the SynLZO algorithm</td>
<td>1395</td>
</tr>
<tr>
<td><code>lzopas_compress</code></td>
<td>Compress in_p(in_len) into out_p</td>
<td>1395</td>
</tr>
<tr>
<td><code>lzopas_compressdestlen</code></td>
<td>Get maximum possible (worse) compressed size for out_p</td>
<td>1395</td>
</tr>
<tr>
<td><code>lzopas_decompress</code></td>
<td>Uncompress in_p(in_len) into out_p (must be allocated before call), returns out_len</td>
<td>1395</td>
</tr>
<tr>
<td><code>lzopas_decompressdestlen</code></td>
<td>Get uncompressed size from lzo-compressed buffer (to reserve memory, e.g.)</td>
<td>1395</td>
</tr>
</tbody>
</table>

```pascal
function CompressSynLZO(var Data: AnsiString; Compress: boolean): AnsiString;
(\textit{(de)}compress a data content using the SynLZO algorithm)
\text{- as expected by THttpSocket.RegisterCompress}
\text{- will return 'synlzo' as ACCEPT-ENCODING: header parameter}
\text{- will store a hash of both compressed and uncompressed stream: if the data is corrupted during transmission, will instantly return '"

function lzopas_compress(in_p: PAnsiChar; in_len: integer; out_p: PAnsiChar): integer;
\textit{Compress in}_p(\textit{in}_\textit{len}) \textit{into out}_p
\text{- out}_p \text{ must be at least} \text{lzopas}_\text{compressdestlen}(\text{in}_\text{len}) \text{ bytes long}
\text{- returns compressed size in out}_p

function lzopas_compressdestlen(in_len: integer): integer;
\textit{Get maximum possible (worse) compressed size for out}_p

function lzopas_decompress(in_p: PAnsiChar; in_len: integer; out_p: PAnsiChar): Integer;
\textit{Uncompress in}_p(\textit{in}_\textit{len}) \textit{into out}_p (\textit{must be allocated before call}), \textit{returns out}_len
\text{- may write up to out}_\text{len}+3 \text{ bytes in out}_p
\text{- the decompression mode is "fast-unsafe" -> CRC/Adler32 in}_p \text{ data before call}

function lzopas_decompressdestlen(in_p: PAnsiChar): integer;
\textit{Get uncompressed size from lzo-compressed buffer (to reserve memory, e.g.)}
```
27.26. SynMongoDB.pas unit

*Purpose*: MongoDB document-oriented database direct access classes
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the *SynMongoDB* unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrtSock</td>
<td>Classes implementing TCP/UDP/HTTP client and server protocol</td>
<td>1083</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
<tr>
<td></td>
<td>- implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>
SynMongoDB class hierarchy

Objects implemented in the SynMongoDB unit

| Objects                      | Description                                                          | Page |
|------------------------------|                                                                     |      |
| EBSONException               | Exception type used for BSON process                                 | 1401 |
| EMongoConnectionException    | Exception type used for MongoDB process, once connected              | 1435 |
| EMongoException              | Exception type used for MongoDB process                               | 1411 |
| EMongoRequestException       | Exception type used for MongoDB query process                         | 1435 |
| EMongoRequestOSException     | Exception type used for MongoDB query process after an Operating System error (e.g. in case of socket error) | 1436 |
| TBSON24                      | 24-bit storage, mapped as a 3 bytes buffer                           | 1401 |
| TBSONElement                 | Data structure used during BSON binary decoding of one BSON element    | 1404 |
| TBSONIterator                | Data structure used for iterating over a BSON binary buffer           | 1407 |
| TBSONObjectID                | BSON ObjectID 12-byte internal binary representation                   | 1401 |
| TBSONVariant                 | Custom variant type used to store some special BSON elements          | 1403 |
| TBSONVariantData             | Memory structure used for some special BSON storage as variant         | 1402 |
| TBSONWriter                  | Used to write the BSON context                                       | 1407 |
### Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDecimal128</td>
<td>Handles a 128-bit decimal value</td>
<td>1399</td>
</tr>
<tr>
<td>TDecimal128Bits</td>
<td>Binary representation of a 128-bit decimal, stored as 16 bytes</td>
<td>1398</td>
</tr>
<tr>
<td>TMongoClient</td>
<td>Remote access to a MongoDB server</td>
<td>1422</td>
</tr>
<tr>
<td>TMongoCollection</td>
<td>Remote access to a MongoDB collection</td>
<td>1426</td>
</tr>
<tr>
<td>TMongoConnection</td>
<td>One TCP/IP connection to a MongoDB server</td>
<td>1419</td>
</tr>
<tr>
<td>TMongoDatabase</td>
<td>Remote access to a MongoDB database</td>
<td>1425</td>
</tr>
<tr>
<td>TMongoReplyCursor</td>
<td>Map a MongoDB server reply message as sent by the database</td>
<td>1416</td>
</tr>
<tr>
<td>TMongoReplyHeader</td>
<td>Internal low-level binary structure mapping the TMongoReply header</td>
<td>1415</td>
</tr>
<tr>
<td>TMongoRequest</td>
<td>Abstract class used to create MongoDB Wire Protocol client messages</td>
<td>1411</td>
</tr>
<tr>
<td>TMongoRequestDelete</td>
<td>A MongoDB client message to delete one or more documents in a collection</td>
<td>1413</td>
</tr>
<tr>
<td>TMongoRequestGetMore</td>
<td>A MongoDB client message to continue the query of one or more documents in a collection, after a TMongoRequestQuery message</td>
<td>1414</td>
</tr>
<tr>
<td>TMongoRequestInsert</td>
<td>A MongoDB client message to insert one or more documents in a collection</td>
<td>1412</td>
</tr>
<tr>
<td>TMongoRequestKillCursors</td>
<td>A MongoDB client message to close one or more active cursors</td>
<td>1415</td>
</tr>
<tr>
<td>TMongoRequestQuery</td>
<td>A MongoDB client message to query one or more documents in a collection</td>
<td>1413</td>
</tr>
<tr>
<td>TMongoRequestUpdate</td>
<td>A MongoDB client message to update a document in a collection</td>
<td>1412</td>
</tr>
<tr>
<td>TMongoRequestWritable</td>
<td>A MongoDB client abstract ancestor which is able to create a BULK command message for MongoDB &gt;= 2.6 instead of older dedicated Wire messages</td>
<td>1412</td>
</tr>
<tr>
<td>TMongoWireHeader</td>
<td>Internal low-level binary structure mapping all message headers</td>
<td>1415</td>
</tr>
</tbody>
</table>

**TDecimal128Bits = record**

*Binary representation of a 128-bit decimal, stored as 16 bytes*

- i.e. IEEE 754-2008 128-bit decimal floating point as used in the BSON Decimal128 format, and processed by the TDecimal128 object
**TDecimal128 = object(TObject)**

*Handles a 128-bit decimal value*
- i.e. IEEE 754-2008 128-bit decimal floating point as used in the BSON Decimal128 format, i.e. `betDecimal128` `TBSONElementType`
- the `betFloat` BSON format stores a 64-bit floating point value, which doesn't have exact decimals, so may suffer from rounding or approximation
- for instance, if you work with Delphi currency values, you may store `betDecimal128` values in MongoDB - the easiest way is to include it as a `TBSONVariant` instance, via the `NumberDecimal()` function
- there is no mathematical operator/methods for Decimal128 Value Objects, as required by MongoDB specifications: any computation must be done explicitly on native language value representation (e.g. currency, `TBCD` or any `BigNumber` library) - use `ToCurr/FromCurr` or `ToText/FromText` to make the appropriate safe conversions

**Bits: TDecimal128Bits;**
*The raw binary storage*

**function Equals(const other: TDecimal128): boolean;**
*Fast bit-per-bit value comparison*

**function FromFloat(const value: TSynExtended; precision: integer=0): boolean;**
*Fills with a native floating-point value*
- note that it doesn't make much sense to use this method: you should rather use the native `betFloat` BSON format, with native double precision
- this method is just a wrapper around `ExtendedToShort` and `ToText`, so you should provide the expected precision, from the actual storage variable (you may specify e.g. `SINGLE_PRECISION` or `EXTENDED_PRECISION` if you don't use a double kind of value)

**function FromText(const text: RawUTF8): TDecimal128SpecialValue; overload;**
*Fills from the text representation of a decimal value*
- returns `dsvValue` or one of the `dsvNan`, `dsvZero`, `dsvPosInf`, `dsvNegInf` special value indicator otherwise on success
- returns `dsvError` on parsing failure

**function FromText(const text: PUTF8Char; textlen: integer): TDecimal128SpecialValue; overload;**
*Fills from the text representation of a decimal value*
- returns `dsvValue` or one of the `dsvNan`, `dsvZero`, `dsvPosInf`, `dsvNegInf` special value indicator otherwise on success
- returns `dsvError` on parsing failure

**function FromVariant(const value: variant): boolean;**
*Convert a variant into one Decimal128 value*
- will first check for a `TBSONVariant` containing a `betDecimal128` (e.g. as retrieved via the `ToVariant` method)
- will recognize currency and `VariantToInt64()` stored values
- then will try to convert the variant from its string value, expecting a floating-point text content
- returns TRUE if conversion was made, FALSE on any error
function IsSpecial: TDecimal128SpecialValue;

Checks if the value matches one of the known special values
- will search for dsvNan, dsvZero, dsvPosInf, dsvNegInf, dsvMin, dsvMax

function ToCurr: currency; overload;

Converts this Decimal128 value to a fixed decimal value
- by design, some information may be lost during conversion, unless the value has been stored
  previously via the FromCurr() method - in this case, conversion is immediate and accurate

function ToFloat: TSynExtended;

Converts this Decimal128 value to a floating-point value
- by design, some information may be lost during conversion
- note that it doesn't make much sense to use this method: you should rather use the native
  betFloat BSON format, with native double precision

function ToText(out Buffer: TDecimal128Str): integer; overload;

Converts the value to its string representation
- returns the number of AnsiChar written to Buffer

function ToText: RawUTF8; overload;

Converts this Decimal128 value to its string representation

function ToVariant: variant; overload;

Convert this Decimal128 value to its TBSONVariant custom variant value

procedure AddText(W: TTextWriter);

Converts this Decimal128 value to its string representation

procedure FromCurr(const value: Currency);

Fills with a fixed decimal value, as stored in currency
- will store the content with explicitly four decimals, as in currency
- by design, this method is very fast and accurate

procedure FromInt32(value: integer);

Fills with a 32-bit signed value

procedure FromInt64(value: Int64);

Fills with a 64-bit signed value

procedure FromQWord(value: QWord);

Fills with a 64-bit unsigned value

procedure FromUInt32(value: cardinal);

Fills with a 32-bit unsigned value

procedure SetSpecial(special: TDecimal128SpecialValue);

Fills with a special value
- dsvError or dsvValue will set dsvNan binary content

procedure SetZero;

Fills with the Zero value
- note: under IEEE 754, Zero can have sign and exponents, so is not Hi=Lo=0
- is the same as Fill(dsvZero)
procedure ToCurr(out result: currency); overload;

 Converts this Decimal128 value to a fixed decimal value
 - by design, some information may be lost during conversion, unless the value has been stored
   previously via the FromCurr() method - in this case, conversion is immediate and accurate

procedure ToText(var result: RawUTF8); overload;

 Converts this Decimal128 value to its string representation

procedure ToVariant(out result: variant); overload;

 Convert this Decimal128 value to its TBSONVariant custom variant value

EBSONException = class(ESynException)

 Exception type used for BSON process

TBSON24 = record

 24-bit storage, mapped as a 3 bytes buffer
 - as used for TBSONObjectID.MachineID and TBSONObjectID.Counter

TBSONObjectID = object(TObject)

 BSON ObjectID 12-byte internal binary representation
 - in MongoDB, documents stored in a collection require a unique _id field that acts as a primary
   key: by default, it uses such a 12-byte ObjectID
 - by design, sorting by _id: ObjectID is roughly equivalent to sorting by creation time, so ease
   sharding and BTREE storage
 - in our ODM, we rather use 64-bit genuine integer identifiers (TID), as computed by an internal
   sequence or TSynUniqueIdentifierGenerator
   - match betObjectID TBSONElementType

  Counter: TBSON24;
   3-byte counter, starting with a random value
   - used to avoid collision

  MachineID: TBSON24;
   3-byte machine identifier
   - ComputeNew will use a hash of ExeVersion.Host and ExeVersion.User

  ProcessID: word;
   2-byte process id
   - ComputeNew will derivate it from MainThreadID

  UnixCreateTime: cardinal;
   Big-endian 4-byte value representing the seconds since the Unix epoch
   - time is expressed in Coordinated Universal Time (UTC), not local time

function CreateDateTime: TDateTime;

 Returns the timestamp portion of the ObjectId() object as a Delphi date
 - time is expressed in Coordinated Universal Time (UTC), not local time so you can compare it to
   NowUTC returned time

function Equal(const Another: TBSONObjectID): boolean; overload;

 Compare two Object IDs
function Equal(const Another: variant): boolean; overload;
  Compare two Object IDs, the second being stored in a TBSONVariant

function FromText(Text: PUTF8Char): boolean; overload;
  Convert an hexadecimal string value into one ObjectID
  - returns TRUE if conversion was made, FALSE on any error

function FromText(const Text: RawUTF8): boolean; overload;
  Convert an hexadecimal string value into one ObjectID
  - returns TRUE if conversion was made, FALSE on any error

function FromVariant(const value: variant): boolean;
  Convert a variant into one ObjectID
  - will first check for a TBSONVariant containing a betObjectID
  - then will try to convert the variant from its string value, expecting an hexadecimal text content
  - returns TRUE if conversion was made, FALSE on any error

function ToText: RawUTF8; overload;
  Convert this ObjectID to its hexadecimal string value

function ToVariant: variant; overload;
  Convert this ObjectID to its TBSONVariant custom variant value

procedure ComputeNew;
  ObjectId content be filled with some unique values
  - this implementation is thread-safe

procedure Init;
  Set all internal fields to zero

procedure ToText(var result: RawUTF8); overload;
  Convert this ObjectID to its hexadecimal string value

procedure ToVariant(var result: variant); overload;
  Convert this ObjectID to its TBSONVariant custom variant value

TBSONVariantData = packed record
  Memory structure used for some special BSON storage as variant
  - betObjectID kind will store a TBSONObjectID
  - betBinary kind will store a BLOB content as RawByteString
  - betDoc and betArray kind will store a BSON document, in its original binary format as RawByteString (TBSONDocument)
  - betDepreciatedDbptr, betJSScope, betTimestamp and betRegEx will store the raw original BSON content as RawByteString
  - betJS and betDepreciatedSymbol will store the UTF-8 encoded string as a RawUTF8
  - betDepreciatedUndefined or betMinKey/betMaxKey do not contain any data
  - betDecimal128 will store the TDecimal128 16 bytes binary buffer
  - warning: VBlob/VText use should match BSON_ELEMENTVARIANTMANAGED constant
VBlob: pointer;

Store the raw binary content as a RawByteString (or TBSONDocument for betDoc/betArray, i.e. the "int32 e_slot #0" standard layout)
- you have to use RawByteString(VBlob) when accessing this field
- e.g. for betRegEx, it will contain raw [cstring cstring] content

VFiller: array[1..SizeOf(TVarData)-SizeOf(TVarType)-SizeOf(TBSONElementType)-SizeOf(TBSONObjectID)] of byte;

Does not complain if Filler is declared but never used

VKind: TBSONElementType

The kind of element stored

VText: pointer;

Store here a RawUTF8 with the associated text
- you have to use RawUTF8(VText) when accessing this field

VType: TVarType;

The variant type

TBSONVariant = class(TSynInvokeableVariantType)

Custom variant type used to store some special BSON elements
- internal layout will follow TBSONVariantData
- handled kind of item are complex BSON types, like betObjectID, betBinary or betDoc/betArray
- it will allow conversion to/from string (and to date for ObjectID)

function IsOfKind(const V: variant; Kind: TBSONElementType): boolean;

Returns TRUE if the supplied variant stores the supplied BSON kind of value

function ToBlob(const V: Variant; var Blob: RawByteString): boolean;

Retrieve a betBinary content stored in a TBSONVariant instance
- returns TRUE if the supplied variant is a betBinary, and set the binary value into the supplied Blob variable
- returns FALSE otherwise

function TryJSONToVariant(var JSON: PUTF8Char; var Value: variant; EndOfObject: PUTF8Char): boolean; override;

Customization of JSON conversion into TBSONVariant kind of variants

procedure Cast(var Dest: TVarData; const Source: TVarData); override;

Handle type conversion
- only types processed by now are string/OleStr/UnicodeString/date

procedure CastTo(var Dest: TVarData; const Source: TVarData; const AVarType: TVarType); override;

Handle type conversion
- only types processed by now are string/OleStr/UnicodeString/date

procedure Clear(var V: TVarData); override;

Clear the instance
procedure Compare(const Left, Right: TVarData; var Relationship: TVarCompareResult); override;

Compare two variant values
- handle comparison of any variant, including TBSONVariant, via a temporary JSON conversion, and case-sensitive comparison
- it uses case-sensitive text (hexadecimal) comparison for betObjectID

procedure Copy(var Dest: TVarData; const Source: TVarData; const Indirect: Boolean); override;

Copy one instance

procedure FromBinary(const Bin: RawByteString; BinType: TBSONElementBinaryType; var result: variant); override;

Convert a BLOB binary content into a TBSONVariant of kind betBinary
- if Bin is '', will store a NULL variant

procedure FromBSONDocument(const BSONDoc: TBSONDocument; var result: variant; Kind::TBSONElementType=doc); override;

Convert a TBSONDocument binary content into a TBSONVariant of kind betDoc or betArray
- see also all BSONVariant() overloaded functions, which also create a TBSONVariant betDoc instance

procedure FromJSON(json: PUTF8Char; var result: variant); override;

Convert a JSON content into a TBSONVariant of kind betDoc or betArray
- warning: the supplied JSON buffer will be modified in-place
- will create a plain variant value if the JSON doesn't start with [ or {

procedure ToJSON(W: TTextWriter; const Value: variant; Escape: TTextWriterKind); override;

Variant serialization will use modMongoStrict JSON-compatible mode

property NewDoc[const BSONDoc: TBSONDocument]: variant read GetNewDoc;

Convert a TBSONDocument binary content into a TBSONVariant of kind betDoc
- is the default property, so that you can write:
  ```pascal
  BSONVariantType[BSO
  nDoc(['BSON', _ Arr(['awesome', 5.05, 1986])])]
  ```
- see also all BSONVariant() overloaded functions, which also create a TBSONVariant betDoc instance

TBSONElement = object(TObject)

Data structure used during BSON binary decoding of one BSON element
- will be retrieved by FromVariant() or FromNext()
- see http://bsonspec.org/#/specification
- this structure has been optimized to map the BSON binary content, without any temporary memory allocation (the SAX way)

Data: record

Depending on the Kind, will point to parsed complex sub-data
- since variable records can't have properties, we nest this information within this main Data variable record
- not all Kind are handled here, only any complex data
Element: pointer;

Pointer to the BSON element value
- is the raw value, without any parsing, e.g. points to a double value or a document: "int32 e_list #0" standard layout (same as TBSONDocument)
- you may cast it for simple types:
PDouble(Element)^  PBoolean(Element)^  PInteger(Element)^  PInt64(Element)^  PBSONObjectID(Element)^  PDecimal128(Element)^
- or use the nested Data variant record to access more complex content
  warning: equals nil for betString/betJS after FromVariant()

ElementBytes: integer;

Number of bytes in the BSON element
- will include the trailing #0 for string element

Index: integer;

Index of this element in the original sequence list
- is correct only when the element has been reset before the parsing loop, e.g.:
  item.Index := -1;  // so item.Index will count starting at 0
  while item.FromNext(elem.Document) do
    writeln(item.Index, ' ', Item.Name, ' ', Item.ValueBytes);

Kind: TBSONElementType;

The element type

Name: PUTF8Char;

The UTF-8 encoded name of this element

NameLen: integer;

The name length (in chars) of this element

function DocItemToInteger(const aName: RawUTF8; const default: Int64=0): Int64;

Search an integer property value within the BSON element as document
- returns the value if aName has been found as property in the BSON element
- returns default if aName was not found, or if Kind is not betDoc or betArray

function DocItemToRawUTF8(const aName: RawUTF8): RawUTF8;

Search an UTF-8 property value within the BSON element as document
- returns the value if aName has been found as property in the BSON element
- returns "" if aName was not found, or if Kind is not betDoc or betArray

function DocItemToVariant(const aName: RawUTF8; var aValue: variant; DocArrayConversion: TBSONDocArrayConversion=asBSONVariant): boolean;

Search a variant property value within the BSON element as document
- returns true if aName has been found as property in the BSON element, and fills aValue with the corresponding value
- returns false if aName was not found, or if Kind is not betDoc or betArray
function FromDocument(const doc: TBSOneDocument): boolean;

Fill a BSON Element structure from a BSON document
- will check the document length then set Kind := betDoc and Data.DocList
- will return TRUE if the supplied doc has a valid length, FALSE otherwise
- you can later on use DocItemToVariant, DocItemToRawUTF8 or DocItemToInteger methods
- the supplied "doc" variable should remain available until you are done with this TBSOneElement wrapper

function FromNext(var BSON: PByte): boolean;

Fill a BSON Element structure from a BSON encoded binary buffer list
- parse the next BSON element: BSON parameter should point to the "e_list" of the "int32 e_list #0" BSON document
- will decode the supplied binary buffer into the BSON element structure, then it will let BSON point to the next element, and return TRUE
- returns FALSE when you reached betEOF, so that you can use it in a loop, and retrieve all the content as consecutive events, without any memory allocation (the SAX way):
  var bson: PByte;
   item: TBSOneElement;
...
BSONParseLength(bson);
while item.FromNext(bson) do
  writeln(item.Name);
- will raise an EBSONException if BSON content is not correct
- as an alternative, consider using TBSONIterator, which wrap both a PByte and a TBSOneElement into one convenient item

function FromSearch(BSON: PByte; const aName: RawUTF8): boolean;

Search for a given name in a BSON encoded binary buffer list
- BSON parameter should point to the first "e_list" item of the "int32 e_list #0" BSON document
- returns false if the item was not found (with case-insensitive search)
- otherwise returns TRUE and the matching element content has been decoded within this TBSOneElement structure

function ToInteger(const default: Int64=0): Int64;

Convert a BSON element into an integer value
- will work only for betBoolean/betInt32/betInt64 types
- any other kind of values will return the supplied default value

function ToRawUTF8: RawUTF8;

Convert a BSON element into an UTF-8 string
- any complex types (e.g. nested documents) will be converted via a variant

function ToVariant(DocArrayConversion: TBSOneDocArrayConversion=asBSONVariant): variant; overload;

Convert a BSON element, as retrieved by TBSOneElement.FromNext(), into a variant
- it will return either standard variant values, or TBSONVariant custom type for most complex kind of elements (see TBSONVariantData type definition)
- note that betString types will be stored as RawUTF8 varString
- by default, it will return TBSONVariant custom variants for documents or arrays - but if storeDocArrayAsDocVariant is set, it will return a TDocVariant custom kind of variant, able to access to its nested properties via late-binding
procedure AddMongoJSON(W: TTextWriter; Mode: TMongoJSONMode=modMongoStrict);

Convert a BSON element, as retrieved by TBSONElement.FromNext(), into its JSON representation
- this method will use by default the MongoDB Extended JSON syntax for specific MongoDB objects but you may use modMongoShell if needed
- will raise an EBSONException if element is not correct

procedure FromVariant(const aName: RawUTF8; const aValue: Variant; var aTemp: RawByteString);

Fill a BSON Element structure from a variant content and associated name
- perform the reverse conversion as made with ToVariant()
- since the result won't store any data but points to the original binary content, the supplied Name/Value instances should remain available as long as you will access to the result content
- aName here is just for convieniency, and could be left void
- supplied aTemp variable will be used for temporary storage, private to this initialized TBSONElement

procedure ToVariant(var result: variant; DocArrayConversion: TBSONDocArrayConversion=asBSONVariant); overload;

Convert a BSON element, as retrieved by TBSONElement.FromNext(), into a variant
- same as the other ToVariant() overloaded function, but avoiding a copy of the resulting variant

TBSONIterator = object(TObject)
Data structure used for iterating over a BSON binary buffer
- is just a wrapper around a PByte value, to be used with a TBSONDocument

Item: TBSONElement;

Map the current item, after the Next method did return TRUE
- map the global document, after Init() but before the first Next call

function Init(const doc: TBSONDocument; kind: TBSONElementType=betArray): boolean;
Initialize the iteration on the supplied BSON document
- will check the document length and returns TRUE if it is correct
- only accepted kind are betDoc and betArray (but not used later)
- you can then use the Next method to iterate over the Item elements
- after this call, the Item property map to the global BSON document (note that after any call to the Next method, Item will map the current iterated value, and not the global BSON document any more)

function Next: boolean;

Will iterate on the BSON document
- returns TRUE if the item has been retrieved into the Item property
- returns FALSE if we reached the end of the supplied BSON buffer

TBSONWriter = class(TFileBufferWriter)
Used to write the BSON context
function BSONWriteDocFromJSON(JSON: PUTF8Char; aEndOfObject: PUTF8Char; out Kind: TBSONElementType; DoNotTryExtendedMongoSyntax: boolean=false): PUTF8Char;

Write some BSON document from a supplied (extended) JSON array or object
- warning: the incoming JSON buffer will be modified in-place: so you should make a private copy before running this method (see e.g. TSynTempBuffer)
- will handle only '{ ... }', '[ ... ]' or 'null' input, with the standard strict JSON format, or BSON-like extensions, e.g. unquoted field names:
  
  `{id:10,doc:{name:"John",birthyear:1972}}`

- if DoNotTryExtendedMongoSyntax is default FALSE, then the MongoDB Shell syntax will also be recognized to create BSON custom types, like

  `new Date()   ObjectId()   MinKey   MaxKey  /<jRegex>/<jOptions>`

  see @http://docs.mongodb.org/manual/reference/mongodb-extended-json

  `{id:new ObjectId(),doc:{name:"John",date:ISODate()}}`

  `{name:"John",field:/acme.*corp/i}`

- if DoNotTryExtendedMongoSyntax is TRUE, process may be slightly faster
- will create the BSON binary without any temporary TDocVariant storage

function BSONWriteQueryOperator(name: RawUTF8; inverted: boolean; operator: TSynTableStatementOperator; const Value: variant): boolean;

Write an object as query parameter
- will handle all SQL operators, including IN (), IS NULL or LIKE
- see @http://docs.mongodb.org/manual/reference/operator/query
- inverted should be TRUE e.g. for a NOT ... expression
- returns TRUE on success, FALSE if the operator is not implemented yet

procedure BSONAdjustDocumentsSize(BSON: PByteArray); virtual;

After all content has been written, call this method on the resulting memory buffer to store all document size as expected by the standard

procedure BSONDocumentBegin(const name: RawUTF8; kind: TBSONElementType=betDoc);

To be called before a BSON document will be written
- each BSONDocumentBegin should be followed by its nested BSONDocumentEnd
- you could create a new BSON object by specifying a name and its type, i.e. either betDoc or betArray

procedure BSONDocumentBegin; overload;

To be called before a BSON document will be written
- each BSONDocumentBegin should be followed by its nested BSONDocumentEnd

procedure BSONDocumentBeginInArray(const name: RawUTF8; kind: TBSONElementType=betDoc);

To be called before a BSON document will be written in an array
- only one level of array should be used per TBSONWriter class

procedure BSONDocumentEnd(CloseNumber: integer=1; WriteEndingZero: boolean=true);

To be called when a BSON document has been written
- it will store the current stream position into an internal array, which will be written when you call AdjustDocumentsSize()
- you can optional specify how many nested documents should be closed, and/or if it should not write an ending betEof item
procedure BSONWrite(const name: RawUTF8; const bson: TBSONVariantData); overload;
  
  Write a BSONVariant instance value

procedure BSONWrite(const name: RawUTF8; const elem: TBSONElement); overload;
  
  Write an element with no value

procedure BSONWrite(const name: RawUTF8; const doc: TDocVariantData); overload;
  
  Write a DocVariant instance value

procedure BSONWrite(const name: RawUTF8; const value: TVarRec); overload;
  
  Write an open array (const Args: array of const) argument
  - handle simple types (numbers, strings...) and custom types (TDocVariant)

procedure BSONWrite(const name: RawUTF8; const value: TDecimal128); overload;
  
  Write a TDecimal128 value

procedure BSONWrite(const name: RawUTF8; elemtype: TBSONElementType); overload;
  
  Write an element with no value
  - elemType can be either betNull, betMinKey or betMaxKey

procedure BSONWrite(const name: RawUTF8; const value: Int64); overload;
  
  Write a 64 bit integer value

procedure BSONWrite(const name: RawUTF8; value: PUTF8Char); overload;
  
  Write a string (UTF-8) value from a memory buffer

procedure BSONWrite(const name: RawUTF8; const value: RawUTF8; isJavaScript: boolean=false); overload;
  
  Write a string (UTF-8) value

procedure BSONWrite(const name: RawUTF8; const value: integer); overload;
  
  Write a 32 bit integer value

procedure BSONWrite(const name: RawUTF8; const value: Double); overload;
  
  Write a floating point value

procedure BSONWrite(const name: RawUTF8; const value: boolean); overload;
  
  Write a boolean value

procedure BSONWrite(const name: RawUTF8; const value: TBSONObjectID); overload;
  
  Write an ObjectID value

procedure BSONWrite(const name: RawUTF8; Data: pointer; DataLen: integer); overload;
  
  Write a binary (BLOB) value

procedure BSONWriteArray(const Items: array of const); overload;
  
  Write an array specified as a list of items as a BSON document
  - data must be supplied as a list of values e.g.
    aBSONWriter.BSONWriteArray(['John',1972]);
  - this method will be faster than using a BSONWriteDoc(_ArrFast(...))

procedure BSONWriteArray(const kind: TBSONElementType); overload;
  
  Write one array item, i.e. the ASCII index name as text
  - only one level of array should be used per BSONWriter class
procedure BSONWriteArrayOfInt64(const Integers: array of Int64);
Write an array of integers as a BSON Document

procedure BSONWriteArrayOfInteger(const Integers: array of integer);
Write an array of integers as a BSON Document

procedure BSONWriteDateTime(const name: RawUTF8; const value: TDateTime);
Write a data/time value

procedure BSONWriteDoc(const doc: TDocVariantData);
Recursive writing of a BSON document or value from a TDocVariant object or array, used e.g. by
BSON(const doc: TDocVariantData) function
- caller should execute BSONAdjustDocumentsSize() on the resulting buffer
- this method will call BSONDocumentBegin/BSONDocumentEnd internally
- will raise an EBSONException if doc is not a valid TDocVariant or null or if the resulting binary
content is bigger than BSON_MAXDOCUMENTSIZE

procedure BSONWriteFromJSON(const name: RawUTF8; var JSON: PUTF8Char; EndOfObject: PUTF8Char; DoNotTryExtendedMongoSyntax: boolean=false);
Write a value from the supplied JSON content
- is able to handle any kind of values, including nested documents or BSON extended syntax (if
DoNotTryExtendedMongoSyntax=false)
- this method is used recursively by BSONWriteDocFromJSON(), and should not be called directly
- will return JSON=nil in case of unexpected error in the supplied JSON

procedure BSONWriteObject(const NameValuePairs: array of const);
Write an object specified as name/value pairs as a BSON document
- data must be supplied two by two, as Name,Value pairs, e.g.
  aBSONWriter.BSONWriteObject([\'name\',\'John\',\'year\',1972]);
- this method will be faster than using a BSONWriteDoc(_ObjFast(...))

procedure BSONWriteProjection(const FieldNamesCSV: RawUTF8);
Write a projection specified as fieldname:1 pairs as a BSON document

procedure BSONWriteRegEx(const name: RawUTF8; const RegEx,Options: RawByteString);
Write a RegEx value

procedure BSONWriteString(const name: RawUTF8; value: PUTF8Char; valueLen: integer);
Write a string (UTF-8) value from a memory buffer

procedure BSONWriteVariant(const name: RawUTF8; const value: variant); overload;
Write a variant value
- handle simple types (numbers, strings...) and custom types (TDocVariant and TBSONVariant,
trying a translation to JSON for other custom types)

procedure CancelAll; override;
Rewind the Stream to the position when Create() was called
- this will also reset the internal document offset table
**procedure** ToBSONDocument(`var result: TBSONDocument`); **virtual**

Flush the content and return the whole binary encoded stream
- call BSONAdjustDocumentsSize() to adjust all internal document sizes
- expect the TBSONWriter instance to have been created as such:

```pascal
tbsonWriter.Create(TRawByteStringStream);
```

**procedure** ToBSONVariant(`var result: variant; Kind: TBSONElementType=btDoc`);

Flush the content and return the whole document as a TBSONVariant
- call ToBSONDocument() to adjust all internal document sizes
- expect the TBSONWriter instance to have been created as such:

```pascal
tbsonWriter.Create(TRawByteStringStream);
```

EMongoException = **class**(ESynException)

Exception type used for MongoDB process

TMongoRequest = **class**(TBSONWriter)

Abstract class used to create MongoDB Wire Protocol client messages
- see [http://docs.mongodb.org/mongodb-wire-protocol](http://docs.mongodb.org/mongodb-wire-protocol)
- this class is not tight to the connection class itself (which is one known limitation of TMongoWire for instance)

**constructor** Create(`const FullCollectionName: RawUTF8; opCode: TMongoOperation; requestID, responseTo: Integer`); **reintroduce**

Write a standard Message Header for MongoDB client
- opCode is the type of the message
- requestID is a client or database-generated identifier that uniquely identifies this message: in case of opQuery or opGetMore messages, it will be sent in the responseTo field from the database
- responseTo is the requestID taken from previous opQuery or opGetMore

**function** ToJSON(`Mode: TMongoJSONMode`): RawUTF8; overload

Write the main parameters of the request as JSON

**procedure** BSONWriteParam(`const paramDoc: variant`); **overload**

Append a query parameter as a BSON document
- param can be a TDocVariant, e.g. created with:

```pascal
  _JsonFast('{name: "John", age: \$gt: 21}');
  _JsonFastFrt('{name: ?, age: \$gt: ?}'), [], ['John', 21]);
  _JsonFastFrt('{name: ?, field: /*/i}', ['acme.corp', 'John']);
```
- param can be a TBSONVariant containing a TBSONDocument raw binary block created e.g. from:

```pascal
  bsonVariant(['BSON', _Arr(['awesome', 5.05, 1986])])
  bsonVariantType[bson([ 'BSON', _Arr([ 'awesome', 5.05, 1986 ]) ])]
```
- param is null, it will append a void document
- param is a string, it will be converted as expected by most database commands, e.g.

```pascal
  TMongoRequestQuery.Create('admin.$cmd', 'buildinfo', [], 1)
```

will query `{ buildinfo: 1 }` to the `admin.$cmd` collection, i.e.

```pascal
  admin.$cmd.findOne( { buildinfo: 1 } )
```
procedure ToBSONDocument(var result: TBSONDocument); override;
  Flush the content and return the whole binary encoded stream
  - expect the TBSONWriter instance to have been created with reintroduced Create() specific
    constructors inheriting from this TMongoRequest class
  - this overridden version will adjust the size in the message header

procedure ToJSON(W: TTextWriter; Mode: TMongoJSONMode); overload; virtual;
  Write the main parameters of the request as JSON

property CollectionName: RawUTF8 read fCollectionName;
  The associated full collection name, e.g. 'test'

property DatabaseName: RawUTF8 read fDatabaseName;
  The associated full collection name, e.g. 'db'

property FullCollectionName: RawUTF8 read fFullCollectionName;
  The associated full collection name, e.g. 'db.test'

property MongoRequestID: integer read fRequestID;
  Identify the message, after call to any reintroduced Create() constructor

  The message operation code
  - should be either opUpdate, opInsert, opQuery, opGetMore, opDelete or opKillCursors,
    depending on the TMongoRequest* class instantiated

TMongoRequestWritable = class(TMongoRequest)
  A MongoDB client abstract ancestor which is able to create a BULK command message for
  MongoDB >= 2.6 instead of older dedicated Wire messages

TMongoRequestUpdate = class(TMongoRequestWritable)
  A MongoDB client message to update a document in a collection

constructor Create(const FullCollectionName: RawUTF8; const Selector, Update: variant; Flags: TMongoUpdateFlags=[]); reintroduce;
  Initialize a MongoDB client message to update a document in a collection
  - FullCollectionName is e.g. 'dbname.collectionname'
  - how the update will be processed can be customized via Flags
  - Selector is the BSON document query to select the document, supplied as TDocVariant - i.e.
    created via _JsonFast() or _JsonFastFmt() - or as TBSONVariant - i.e. created via BSONVariant() -
    or null if all documents are to be updated
  - Update is the BSON document specification of the update to perform, supplied as TDocVariant
    or TBSONVariant
  - there is no response to an opUpdate message

procedure ToJSON(W: TTextWriter; Mode: TMongoJSONMode); override;
  Write the main parameters of the request as JSON

TMongoRequestInsert = class(TMongoRequestWritable)
  A MongoDB client message to insert one or more documents in a collection
constructor Create(const FullCollectionName: RawUTF8; const JSONDocuments: array of PUTF8Char; Flags: TMongoInsertFlags=[]); reintroduce; overload;

Initialize a MongoDB client message to insert one or more documents in a collection, supplied as JSON objects
- FullCollectionName is e.g. 'dbname.collectionname'
- JSONDocuments is an array of JSON objects
- there is no response to an opInsert message
- warning: JSONDocuments[] buffer will be modified in-place during parsing, so a private copy may have to be made by the caller

constructor Create(const FullCollectionName: RawUTF8; const Documents: TBSONDocument; Flags: TMongoInsertFlags=[]); reintroduce; overload;

Initialize a MongoDB client message to insert one or more documents in a collection, supplied as raw BSON binary
- FullCollectionName is e.g. 'dbname.collectionname'
- Documents is the low-level concatenation of BSON documents, created e.g. with a TBSONWriter stream
- there is no response to an opInsert message

constructor Create(const FullCollectionName: RawUTF8; const Documents: array of variant; Flags: TMongoInsertFlags=[]); reintroduce; overload;

Initialize a MongoDB client message to insert one or more documents in a collection, supplied as variants
- FullCollectionName is e.g. 'dbname.collectionname'
- Documents is an array of TDocVariant or TBSONVariant - i.e. created via _JsonFast() _JsonFastFmt() or BSONVariant() -
- there is no response to an opInsert message

TMongoRequestDelete = class(TMongoRequestWritable)
A MongoDB client message to delete one or more documents in a collection

constructor Create(const FullCollectionName: RawUTF8; const Selector: variant; Flags: TMongoDeleteFlags=[]); reintroduce;

Initialize a MongoDB client message to delete one or more documents in a collection
- FullCollectionName is e.g. 'dbname.collectionname'
- Selector is the BSON document query to select the document, supplied as TDocVariant - i.e. created via _JsonFast() or _JsonFastFmt() - or as TBSONVariant - i.e. created via BSONVariant() - or null if all documents are to be deleted
- warning: CreateDelete('db.coll',null) can be expensive so you should better drop the whole collection
- there is no response to an opDelete message

procedure ToJSON(W: TTextWriter; Mode: TMongoJSONMode); override;
Write the main parameters of the request as JSON

TMongoRequestQuery = class(TMongoRequest)
A MongoDB client message to query one or more documents in a collection
constructor Create(const FullCollectionName: RawUTF8; const Query, ReturnFieldsSelector: variant; NumberToReturn: integer; NumberToSkip: integer=0; Flags: TMongoQueryFlags=[]); reintroduce;

Initialize a MongoDB client message to query one or more documents in a collection from a specified Cursor identifier
- FullCollectionName is e.g. 'dbname.collectionname'
- Query is the BSON document query to select the document, supplied as TDocVariant - i.e. created via _JsonFast() or _JsonFastFmt() - or null if all documents are to be retrieved - for instance:
  _JsonFast('{name:"John",age:{$gt:21}}');
  _JsonFastFmt('{name:?,age:{$gt:?}}',['John',21]);
  _JsonFastFmt('{name:?,field:/%/i}',['acme.*corp'],['John']);
- if Query is a string, it will be converted as expected by most database commands, e.g.
  TMongoRequestQuery.Create('admin.$cmd','buildinfo',[],1)
  will query { buildinfo: 1 } to the admin.$cmd collection, i.e. admin.$cmd.findOne( { buildinfo: 1 } )
- Query can also be a TBSONVariant, e.g. created with:
  BSONVariant('{name:?,age:{$gt:?}}',['John',21])
- ReturnFieldsSelector is an optional selector (set to null if not applicable) as a BSON document that limits the fields in the returned documents, supplied as TDocVariant or TBSONVariant - e.g. created via:
  BSONVariantFieldSelector('a,b,c');
  BSONVariantFieldSelector(['a','b','c']);
  BSONVariant('{a:1,b:1,c:1}');
  _JsonFast('{a:1,b:1,c:1}');
- if ReturnFieldsSelector is a string, it will be converted into
  { ReturnFieldsSelector: 1 }

procedure ToJSON(W: TTextWriter; Mode: TMongoJSONMode); override;
Write the main parameters of the request as JSON

property NumberToReturn: integer read fNumberToReturn;
Retrieve the NumberToReturn parameter as set to the constructor

property NumberToSkip: integer read fNumberToSkip;
Retrieve the NumberToSkip parameter as set to the constructor

TMongoRequestGetMore = class(TMongoRequest)
A MongoDB client message to continue the query of one or more documents in a collection, after a TMongoRequestQuery message

constructor Create(const FullCollectionName: RawUTF8; NumberToReturn: integer; CursorID: Int64); reintroduce;
Initialize a MongoDB client message to continue the query of one or more documents in a collection, after a opQuery / TMongoRequestQuery message
- FullCollectionName is e.g. 'dbname.collectionname'
- you can specify the number of documents to return (e.g. from previous opQuery response)
- CursorID should have been retrieved within an opReply message from the database
TMongoRequestKillCursor = class(TMongoRequest)
    A MongoDB client message to close one or more active cursors

    constructor Create(const FullCollectionName: RawUTF8; const CursorIDs: array of Int64);
    reintroduce;
    Initialize a MongoDB client message to close one or more active cursors in the database
    - it is mandatory to ensure that database resources are reclaimed by the client at the end of the
      query
    - if a cursor is read until exhausted (read until opQuery or opGetMore returns zero for the
      CursorId), there is no need to kill the cursor
    - there is no response to an opKillCursor message

    procedure ToJSON(W: TTextWriter; Mode: TMongoJSONMode); override;
    Write the main parameters of the request as JSON

TMongoWireHeader = packed record
    Internal low-level binary structure mapping all message headers

    MessageLength: integer;
    Total message length, including the header

    OpCode: integer;
    Low-level code of the message
    - GetReply() will map it to a high-level TMongoOperation

    RequestID: integer;
    Identifier of this message

    ResponseTo: integer;
    Retrieve the RequestID from the original request

TMongoReplyHeader = packed record
    Internal low-level binary structure mapping the TMongoReply header
    - used e.g. by TMongoReplyCursor and TMongoConnection.GetReply()

    CursorID: Int64;
    Cursor identifier if the client may need to perform further opGetMore

    Header: TMongoWireHeader;
    Standard message header

    NumberReturned: integer;
    Number of documents in the reply

    ResponseFlags: integer;
    Response flags

    StartingFrom: integer;
    Where in the cursor this reply is starting
TMongoReplyCursor = object(TObject)

Map a MongoDB server reply message as sent by the database
- in response to TMongoRequestQuery / TMongoRequestGetMore messages
- you can use the record's methods to retrieve information about a given response, and navigate
  within all nested documents
- several TMongoReplyCursor instances may map the same TMongoReply content
- you can safely copy one TMongoReplyCursor instance to another

function AppendAllToDocVariant(var Dest: TDocVariantData): integer;

Append all documents content to a TDocVariant array instance
- if the supplied instance if not already a TDocVariant of kind dvArray, a new void instance will
  be created
- return the new size of the Dest array

function AppendAllToDocVariantDynArray(var Dest: TVariantDynArray): integer;

Append all documents content to a dynamic array of TDocVariant
- return the new size of the Dest[] array

function Next(out JSON: RawUTF8; Mode: TMongoJSONMode=modMongoStrict): boolean; overload;

Retrieve the next document in the list, as JSON content
- return TRUE if the supplied document has been retrieved
- return FALSE if there is no more document to get - you can use the Rewind method to restart
  from the first document
- could be used e.g. as:
  var Reply: TMongoReply;
  json: RawUTF8;
  ...
  Reply.Init(ResponseMessage);
  while Reply.Next(json,modMongoShell) do
    writeln(json); // fast display

function Next(out BSON: TBSONDocument): boolean; overload;

Retrieve the next document in the list, as a BSON binary document
- return TRUE if the supplied document has been retrieved - then doc points to a "int32 e_list
  #0" BSON document
- return FALSE if there is no more document to get - you can use the Rewind method to restart
  from the first document
- this method is slightly slower than the one returning a PByte, since it will allocate a memory
  buffer to store the TBSONDocument binary
- could be used e.g. as:
  var Reply: TMongoReply;
  doc: TBSONDocument;
  ...
  Reply.Init(ResponseMessage);
  while Reply.Next(doc) do
    writeln(BSONToJSON(doc,0,modMongoShell)); // fast display
function Next(out doc: variant; option:
TBSOneDocArrayConversion=asDocVariantPerReference): boolean; overload;

Retrieve the next document in the list, as a TDocVariant instance
- return TRUE if the supplied document has been retrieved
- return FALSE if there is no more document to get - you can use the Rewind method to restart from the first document
- could be used e.g. as:
  var Reply: TMongoReply;
  doc: variant;
  ...
  Reply.Init(ResponseMessage);
  while Reply.Next(doc) do
    writeln('Name: ',doc.Name,' FirstName: ',doc.FirstName);

function Next(out doc: PByte): boolean; overload;

Retrieve the next document in the list, as BSON content
- return TRUE if the supplied document has been retrieved - then doc points to a "int32 e_list #0" BSON document
- return FALSE if there is no more document to get - you can use the Rewind method to restart from the first document
- this method is almost immediate, since the BSON raw binary is returned directly without any conversion
- could be used e.g. as:
  var Reply: TMongoReply;
  doc: PByte;
  ...
  Reply.Init(ResponseMessage);
  while Reply.Next(doc) do
    writeln(BSONToJSON(doc,0,modMongoShell)); // fast display

function ToJSON(Mode: TMongoJSONMode=modMongoStrict; WithHeader: boolean=false; MaxSize: Cardinal=0): RawUTF8;

Return all documents content as a JSON array, or one JSON object if there is only one document in this reply
- this method is very optimized and will convert the BSON binary content directly into JSON

procedure AppendAllToBSON(Dest: TBSONWriter);

Append all documents content to a BSON binary stream
- Dest.Tag will be used to count the current item number in the resulting BSON array

procedure FetchAllToJSON(W: TTextWriter; Mode: TMongoJSONMode=modMongoStrict; WithHeader: boolean=false; MaxSize: Cardinal=0);

Return all documents content as a JSON array, or one JSON object if there is only one document in this reply
- this method is very optimized and will convert the BSON binary content directly into JSON
#### procedure GetDocument(index: integer; var result: variant);

*Retrieve a given document as a TDocVariant instance*
- this method won't use any cache (like Document[..] property), since it should be used with a local variant on stack as cache:

```delphi
begin
  var Reply: TMongoReply;
  doc: variant;
  i: integer;
  ... Reply.Init( HttpResponseMessage);
  for i := 0 to Reply.DocumentCount - 1 do begin
    GmrfQueryFailureGetDocument(i, doc);
    writeln('Name: ',doc.Name, ' FirstName: ',doc.FirstName);
  end;
end;
```

#### procedure Init(const ReplyMessage: TMongoReply);

*Initialize the cursor with a supplied binary reply from the server*
- will raise an EMongoException if the content is not valid
- will populate all record fields with the supplied data

#### procedure Rewind;

*Let Next() overloaded methods point to the first document of this message*

#### property CursorID: Int64 read fCursorID;

*Cusor identifier if the client may need to perform further TMongoRequestGetMore messages*
- in the event that the result set of the query fits into one OP_REPLY message, CursorID will be 0

#### property Document[index: integer]: variant read GetOneDocument;

*Retrieve a given document as a TDocVariant instance*
- could be used e.g. as:

```delphi
begin
  var Reply: TMongoReply;
  i: integer;
  ... Reply.Init( HttpResponseMessage);
  for i := 0 to Reply.DocumentCount - 1 do begin
    writeln('Name: ',Reply.Document[i].Name, ' FirstName: ',Reply.Document[i].FirstName);
  end;
end;
```
- note that there is an internal cache for the latest retrieved document by this property, so that you can call ReplyDOCUMENT[i] several times without any noticeable speed penalty

#### property DocumentBSON: TPointerDynArray read fDocuments;

*Direct access to the low-level BSON binary content of each document*

#### property DocumentCount: Integer read fNumberReturned;

*Number of documents in the reply*

#### property FirstDocument: PAnsiChar read fFirstDocument;

*Points to the first document binary*
- i.e. just after the Reply header

#### property Position: integer read fCurrentPosition;

*The current position of the Next() call, starting at 0*

#### property Reply: TMongoReply read fReply;

*Access to the low-level binary reply message*
<table>
<thead>
<tr>
<th>property</th>
<th>RequestID: integer read fRequestID;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Identifier of this message</td>
</tr>
<tr>
<td>property</td>
<td>ResponseFlags: TMongoReplyCursorFlags read fResponseFlags;</td>
</tr>
<tr>
<td></td>
<td>Retrieve the context execution of this message</td>
</tr>
<tr>
<td>property</td>
<td>ResponseTo: integer read fResponseTo;</td>
</tr>
<tr>
<td></td>
<td>Retrieve the RequestID from the original request</td>
</tr>
<tr>
<td>property</td>
<td>StartingFrom: integer read fStartingFrom;</td>
</tr>
<tr>
<td></td>
<td>Where in the cursor this reply is starting</td>
</tr>
</tbody>
</table>

```
TMongoConnection = class(TObject)

One TCP/IP connection to a MongoDB server
- all access will be protected by a mutex (critical section): it is thread safe but you may use one TMongoClient per thread or a connection pool, for better performance
```

```
constructor Create(const aClient: TMongoClient; const aServerAddress: RawByteString; aServerPort: integer=MONGODB_DEFAULTPORT); reintroduce;

Initialize the connection to the corresponding MongoDB server
- the server address is either a host name, or an IP address
- if no server address is specified, will try to connect to localhost
- this won't create the connection, until Open method is executed
```

```
destructor Destroy; override;

Release the connection, including the socket
```

```
function GetBSONAndFree(Query: TMongoRequestQuery): TBSONDocument;

Send a query to the server, returning a TBSONDocument instance containing all the incoming data, as raw binary BSON document containing an array of the returned items
- will send the Request message, and any needed TMongoRequestGetMore messages to retrieve all the data from the server
- the supplied Query instance will be released when not needed any more
```

```
function GetDocumentsAndFree(Query: TMongoRequestQuery): variant; overload;

Send a query to the server, returning a TDocVariant instance containing all the incoming data
- will send the Request message, and any needed TMongoRequestGetMore messages to retrieve all the data from the server
- the supplied Query instance will be released when not needed any more
- if Query.NumberToReturn<>1, it will return either null or a dvArray kind of TDocVariant containing all returned items
- if Query.NumberToReturn=1, then it will return either null or a single TDocVariant instance
```
function GetJSONAndFree(Query: TMongoRequestQuery; Mode: TMongoJSONMode): RawUTF8;
    Send a query to the server, returning all the incoming data as JSON
    - will send the Request message, and any needed TMongoRequestGetMore messages to retrieve all the data from the server
    - this method is very optimized and will convert the BSON binary content directly into JSON, in either modMongoStrict or modMongoShell layout (modNoMongo will do the same as modMongoStrict)
    - if Query.NumberToReturn<>1, it will return either 'null' or a '[..]' JSON array with all the incoming documents retrieved from the server
    - if Query.NumberToReturn=1, it will return either 'null' or a single '{...}' JSON object
    - the supplied Query instance will be released when not needed any more

function RunCommand(const aDatabaseName: RawUTF8; const command: variant; var returnedValue: variant): RawUTF8; overload;
    Run a database command, supplied as a TDocVariant, TBSONVariant or a string, and return the a TDocVariant instance
    - see http://docs.mongodb.org/manual/reference/command for a list of all available commands
    - for instance:
      RunCommand('test','.DbFast', ['dbStats',1,'scale',1024],stats);
      RunCommand('test',BSONVariant([ 'dbStats',1,'scale',1024],stats);
      RunCommand('admin','buildinfo',fServerBuildInfo);
    - the message will be returned by the server as a single TDocVariant instance (since the associated TMongoRequestQuery.NumberToSkip=1)
    - in case of any error, the error message is returned as text
    - in case of success, this method will return "

function RunCommand(const aDatabaseName: RawUTF8; const command: variant; var returnedValue: TBSONDocument): boolean; overload;
    Run a database command, supplied as a TDocVariant, TBSONVariant or a string, and return the raw BSON document array of received items
    - this overloaded method can be used on huge content to avoid the slower conversion to an array of TDocVariant instances
    - in case of success, this method will return TRUE, or FALSE on error

function SendAndFree(Request: TMongoRequest; NoAcknowledge: boolean=false): variant;
    Send a message to the MongoDB server
    - will apply Client.WriteConcern policy, and run an EMongoException in case of any error
    - the supplied Request instance will be released when not needed any more
    - by default, it will follow Client.WriteConcern pattern - but you can set NoAcknowledge = TRUE to avoid calling the getLastError command
    - will return the getLastError reply (if retrieved from server)

procedure Close;
    Disconnect from MongoDB server
    - will raise an EMongoException on error
procedure GetCursor(Request: TMongoRequest; var Result: TMongoReplyCursor);

Low-level method to send a request to the server, and return a cursor
- if Request is not either TMongoRequestQuery or TMongoRequestGetMore, will raise an EMongoException
- then will parse and return a cursor to the reply message as sent back by the database, with logging if necessary
- raise an EMongoException if mrfQueryFailure flag is set in the reply

procedure GetDocumentsAndFree(Query: TMongoRequestQuery; var result: variant);

Send a query to the server, returning a TDocVariant instance containing all the incoming data
- will send the Request message, and any needed TMongoRequestGetMore messages to retrieve all the data from the server
- the supplied Query instance will be released when not needed any more
- if Query.NumberToReturn<>1, it will return either null or a dvArray kind of TDocVariant containing all returned items
- if Query.NumberToReturn=1, then it will return either null or a single TDocVariant instance

procedure GetDocumentsAndFree(Query: TMongoRequestQuery; var result: TVariantDynArray); overload;

Send a query to the server, returning a dynamic array of TDocVariant instance containing all the incoming data
- will send the Request message, and any needed TMongoRequestGetMore messages to retrieve all the data from the server
- the supplied Query instance will be released when not needed any more

procedure GetRepliesAndFree(Query: TMongoRequestQuery; OnEachReply: TOnMongoConnectionReply; var Opaque);

Low-level method to send a query to the server, calling a callback event on each reply
- is used by GetDocumentsAndFree, GetBSONAndFree and GetJSONAndFree methods to receive the whole document (you should better call those)
- the supplied Query instance will be released when not needed any more

procedure GetReply(Request: TMongoRequest; out result: TMongoReply);

Low-level method to send a request to the server
- if Request is not either TMongoRequestQuery or TMongoRequestGetMore, will raise an EMongoException
- then will return the reply message as sent back by the database, ready to be accessed using a TMongoReplyCursor wrapper

procedure Open;

Connect to the MongoDB server
- will raise an EMongoException on error

property Client: TMongoClient read fClient;

Access to the corresponding MongoDB server

property Locked: boolean read GetLocked;

Is TRUE when the connection is busy

property Opened: boolean read GetOpened;

Return TRUE if the Open method has successfully been called
property ServerAddress: RawUTF8 read fServerAddress;
  Read-only access to the supplied server address
  - the server address is either a host name, or an IP address

property ServerPort: integer read fServerPort;
  Read-only access to the supplied server port
  - the server Port is MONGODB_DEFAULTPORT (27017) by default

property Socket: TCrtSocket read fSocket;
  Direct access to the low-level TCP/IP communication socket

TMongoClient = class(TObject)
  Remote access to a MongoDB server
  - a single server can have several active connections, if some secondary hosts were defined

constructor Create(const Host: RawUTF8; Port: Integer=MONGODB_DEFAULTPORT; aTLS: boolean=false; const SecondaryHostCSV: RawUTF8=''; const SecondaryPortCSV: RawUTF8=''); overload;
  Prepare a connection to a MongoDB server or Replica Set
  - this constructor won't create the connection until the Open method is called
  - you can specify multiple hosts, as CSV values, if necessary
  - depending on the platform, you may request for a TLS secured connection

destructor Destroy; override;
  Close the connection and release all associated TMongoDatabase, TMongoCollection and TMongoConnection instances

function Open(const DatabaseName: RawUTF8): TMongoDatabase;
  Connect to a database on a remote MongoDB primary server
  - this method won't use authentication, and will return the corresponding MongoDB database instance
  - this method is an alias to the Database[] property

function OpenAuth(const DatabaseName,UserName,PassWord: RawUTF8; ForceMongoDBCR: boolean=false): TMongoDatabase;
  Secure connection to a database on a remote MongoDB server
  - this method will use authentication and will return the corresponding MongoDB database instance, with a dedicated secured connection
  - will use MONGODBCR for MongoDB engines up to 2.6 (or if ForceMongoDBCR is TRUE), and SCRAM-SHA-1 since MongoDB 3.x
  - see http://docs.mongodb.org/manual/administration/security/access-control

function ServerBuildInfoText: RawUTF8;
  Retrieve extended server version and build information, as text
  - will create a string from ServerBuildInfo object, e.g. as
  'MongoDB 3.2.8 mozjs mmapv1,wiredTiger'
procedure SetLog(LogClass: TSynLogClass; RequestEvent: TSynLogInfo=sllSQL; 
ReplyEvent: TSynLogInfo=sllDB; ReplyEventMaxSize: cardinal=1024);

- Define an optional logging instance to be used
- you can also specify the event types to be used for requests or replay: by default, a verbose log
- e.g. mORMotMongoDB.pas will call Client.SetLog(SQLite3Log) for you

property BytesReceived: Int64 read GetBytesReceived;
How many bytes this client did received, among all its connections

property BytesSent: Int64 read GetBytesSent;
How many bytes this client did transmitted, among all its connections

property BytesTransmitted: Int64 read GetBytesTransmitted;
How many bytes this client did transmit, adding both input and output

property Connections: TMongoConnectionDynArray read fConnections;
Low-level access to the TCP/IP connections of this MongoDB replica set
- first item [0] is the Primary member
- other items [1..] are the Secondary members

property ConnectionString: RawUTF8 read fConnectionString;
The connection definition used to connect to this MongoDB server

property ConnectionTimeOut: Cardinal read fConnectionTimeOut write fConnectionTimeOut;
The connection time out, in milliseconds
- default value is 30000, i.e. 30 seconds

property ConnectionTLS: boolean read fConnectionTLS;
If the socket connection is secured over TLS

property Database[const DatabaseName: RawUTF8]: TMongoDatabase read Open;
Access to a given MongoDB database
- try to open it via a non-authenticated connection if it not already: will raise an exception on
error, or will return an instance
- will return an existing instance if has already been opened

property GracefulReconnect: boolean read fGracefulReconnect.Enabled write fGracefulReconnect.Enabled;
Allow automatic reconnection (with authentication, if applying), if the socket is closed (e.g. was
dropped from the server)

property Log: TSynLog read fLog write fLog;
Define the logging instance to be used for LogRequestEvent/LogReplyEvent
- you may also call the SetLog() method to set all options at once
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Type</th>
<th>Read Access Method</th>
<th>Write Access Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogReplyEvent</td>
<td>TSynLogInfo</td>
<td>fLogReplyEvent</td>
<td>fLogReplyEvent</td>
<td>If set to something else than default sllNone, will log each reply with the corresponding logging event kind. - WARNING: logging all incoming data may be very verbose, e.g. when retrieving a document list. - use it with care, not on production, but only for debugging purposes - or set LogReplyEventMaxSize to a low value. - will use the Log property for the destination log. - you may also call the SetLog() method to set all options at once.</td>
</tr>
<tr>
<td>LogReplyEventMaxSize</td>
<td>cardinal</td>
<td>fLogReplyEventMaxSize</td>
<td>fLogReplyEventMaxSize</td>
<td>Defines how many characters a LogReplyEvent entry may append in the log - is set by default to 1024, which sounds somewhat good for debugging.</td>
</tr>
<tr>
<td>LogRequestEvent</td>
<td>TSynLogInfo</td>
<td>fLogRequestEvent</td>
<td>fLogRequestEvent</td>
<td>If set to something else than default sllNone, will log each request with the corresponding logging event kind. - will use the Log property for the destination log. - you may also call the SetLog() method to set all options at once.</td>
</tr>
<tr>
<td>ReadPreference</td>
<td>TMongoClientReplicaSetReadPreference</td>
<td>fReadPreference</td>
<td>fReadPreference</td>
<td>Define Read Preference mode to a MongoDB replica set - see <a href="http://docs.mongodb.org/manual/core/read-preference">http://docs.mongodb.org/manual/core/read-preference</a> - default is rpPrimary, i.e. reading from the main primary instance - Important: All read preference modes except rpPrimary may return stale data because secondaries replicate operations from the primary with some delay - ensure that your application can tolerate stale data if you choose to use a non-primary mode.</td>
</tr>
<tr>
<td>ServerBuildInfo</td>
<td>variant</td>
<td>fServerBuildInfo</td>
<td></td>
<td>Retrieve the server version and build information - return the content as a TDocVariant document, e.g. ServerBuildInfo.version = '2.4.9' ServerBuildInfo.versionArray = [2,4,9,0] - this property is cached, so request is sent only once - you may rather use ServerBuildInfoNumber to check for available features at runtime, for easy comparison of the server version.</td>
</tr>
<tr>
<td>ServerBuildInfoNumber</td>
<td>cardinal</td>
<td>fServerBuildInfoNumber</td>
<td></td>
<td>Retrieve the server version and build information - return the content as a TDocVariant document, e.g. 20409000 for MongoDB 2.4.9, or 2860000 for MongoDB 2.6, or 3000300 for MongoDB 3.0.3 - this property is cached, so can be used to check for available features at runtime, without any performance penalty.</td>
</tr>
<tr>
<td>WriteConcern</td>
<td>TMongoClientWriteConcern</td>
<td>fWriteConcern</td>
<td>fWriteConcern</td>
<td>Define Write Concern mode to a MongoDB replica set - see <a href="http://docs.mongodb.org/manual/core/write-concern">http://docs.mongodb.org/manual/core/write-concern</a> - default is wcAcknowledged, i.e. to acknowledge all write operations.</td>
</tr>
</tbody>
</table>
TMongoDatabase = class(TObject)

Remote access to a MongoDB database

constructor Create(aClient: TMongoClient; const aDatabaseName: RawUTF8);

Initialize a reference to a given MongoDB Database
- you should not use this constructor directly, but rather use the TMongoClient.Database[] property
- it will connect to the Client's primary host, then retrieve all collection names of this database

destructor Destroy; override;

Release all associated TMongoCollection instances

function CreateUser(const UserName,Password: RawUTF8; const roles: variant): RawUTF8;

Create the user in the database to which the user will belong
- you could specify the roles to use, for this database or others:

    reportingDB.CreateUser('reportsUser','12345678',BSONVariant(
        '[{ role: "readWrite", db: "reporting" }, { role: "read", db: "products" }]' ));

- returns '' on success, an error message otherwise

function CreateUserForThisDatabase(const UserName,Password: RawUTF8; allowWrite: Boolean=true): RawUTF8;

Create the user with a read or read/write role on the current database
- returns '' on success, an error message otherwise

function DropUser(const UserName: RawUTF8): RawUTF8;

Deletes the supplied user on the current database
- returns '' on success, an error message otherwise

function RunCommand(const command: variant; var returnedValue: variant): RawUTF8;

Run a database command, supplied as a TDocVariant, TBSONVariant or a string, and return a TDocVariant instance
- this is the preferred method to issue database commands, as it provides a consistent interface between the MongoDB shell and this driver
- see http://docs.mongodb.org/manual/reference/command for a list of all available commands
- for instance:

    RunCommand(_ObjFast(['dbStats',1,'scale',1024],stats);
    RunCommand(BSONVariant(['dbStats',1,'scale',1024],stats);
    RunCommand('dbStats',stats);
    RunCommand('hostInfo',host);

- the message will be returned by the server as a TDocVariant instance (since the associated
  TMongoRequestQuery.NumberToSkip=1)
- in case of any error, the error message is returned as text
- in case of success, this method will return "}
function RunCommand(const command: variant; var returnedValue: TBSONDocument): boolean; overload;

    Run a database command, supplied as a TDocVariant, TBSONVariant or a string, and return the raw BSON document array of received items
    - this overloaded method can be used on huge content to avoid the slower conversion to an array of TDocVariant instances
    - in case of success, this method will return TRUE, or FALSE on error

property Client: TMongoClient read fClient;
    The associated MongoDB client instance

property Collection[const Name: RawUTF8]: TMongoCollection read GetCollection;
    Access to a given MongoDB collection
    - raise an EMongoDatabaseException if the collection name does not exist

property CollectionOrCreate[const Name: RawUTF8]:TMongoCollection read GetCollectionOrCreate;
    Access to a given MongoDB collection
    - if the collection name does not exist, it will add use the name to create a TMongoCollection instance and register it to the internal list

property CollectionOrNil[const Name: RawUTF8]: TMongoCollection read GetCollectionOrNil;
    Access to a given MongoDB collection
    - if the collection name does not exist, it will return nil

property Name: RawUTF8 read fName;
    The database name

TMongoCollection = class(TObject)
    Remote access to a MongoDB collection

constructor Create(aDatabase: TMongoDatabase; const aCollectionName: RawUTF8);
    Initialize a reference to a given MongoDB Collection
    - you should not use this constructor directly, but rather use TMongoClient.Database[].Collection[] property

function AggregateDoc(Operators: PUTF8Char; const Params: array of const): variant; overload;
    Calculate aggregate values using the MongoDB aggregation framework and return the result as a TDocVariant instance
    - the Aggregation Framework was designed to be more efficient than the alternative map-reduce pattern, and is available since MongoDB 2.2 - see http://docs.mongodb.org/manual/reference/command/aggregate
    - you should specify the aggregation pipeline as a list of JSON object operators (without the [..]) - for reference of all available phases, see http://docs.mongodb.org/manual/core/aggregation-pipeline
    - if the server sent back no {result:...} member, will return null
    - if the server sent back one item as {result:[..]}, will return this single item as a TDocVariant
    - if the server sent back several items as {result:[..],[..]}, will return a dvArray kind of TDocVariant
function AggregateDocFromJson(const PipelineJSON: RawUTF8): variant;

Calculate aggregate values using the MongoDB aggregation framework and return the result as a TDocVariant instance
- overloaded method to specify the pipeline as a JSON text object as detailed by http://docs.mongodb.org/manual/core/aggregation-pipeline
- for instance, the following will return the maximum _id value of the collection:
  AggregateDoc("{$group:{_id:null,max:{$max:"$_id"}}}"),max

function AggregateDocFromVariant(const pipelineArray: variant): variant;

Calculate aggregate values using the MongoDB aggregation framework and return the result as a TDocVariant instance
- overloaded method to specify the pipeline as a BSON raw document as detailed by http://docs.mongodb.org/manual/core/aggregation-pipeline

function AggregateJSON(Operators: PUTF8Char; const Params: array of const; Mode: TMongoJSONMode=modMongoStrict): RawUTF8; overload;

Calculate JSON aggregate values using the MongoDB aggregation framework
- the Aggregation Framework was designed to be more efficient than the alternative map-reduce pattern, and is available since MongoDB 2.2 - see http://docs.mongodb.org/manual/reference/command/aggregate
- you should specify the aggregation pipeline as a list of JSON object operators (without the [..]) - for reference of all available phases, see http://docs.mongodb.org/manual/core/aggregation-pipeline
- for instance, the following will return as JSON a collection sorted in descending order according by the age field and then in ascending order according to the value in the posts field
  AggregateJSON('{ $sort : { age : -1, posts: 1 } }',[])
function Drop: RawUTF8;

Drops the entire collection from the database
- once dropped, this TMongoCollection instance will be freed: never use this instance again after success (i.e. returned '')
- in case of error, a textual message will be returned as result
- once dropped, this collection will be removed from the parent Database.Collection[] internal list
- Warning: this method obtains a write lock on the affected database and will block other operations until it has completed

function FindBSON(const Criteria, Projection: Variant; NumberToReturn: integer=maxInt; NumberToSkip: Integer=0; Flags: TMongoQueryFlags=[]):
TBSONDocument;

Select documents in a collection and returns a TBSONDocument instance containing the selected documents as a raw binary BSON array document
- Criteria can be null (to retrieve all documents) or a TDocVariant / TBSONVariant query selector:
  FindBSON(BSONVariant({'name':"John",'age':{'$gt':21}}),null);
  FindBSON(BSONVariant({'name':?,'age':{'$gt':?}}),[],['John',21],null);
- Projection can be null (to retrieve all fields) or a CSV string to set the field names to retrieve, or a TDocVariant or TBSONVariant - e.g.:
  FindBSON(BSONVariant(['name','John']),null);
  FindBSON(BSONVariant(['name','John']),'_id');
  FindBSON(BSONVariant(['name','John']),BSONVariantFieldSelector('name,_id'));
- NumberToReturn can be left to its default maxInt value to return all matching documents, or specify a limit (e.g. 1 for one document)

function FindCount(Criteria: PUTF8Char; const Args,Params: array of const; MaxNumberToReturn: integer=0; NumberToSkip: Integer=0): Int64; overload;

Calculate the number of documents in the collection that match a specific query
- Criteria can specify the query selector as (extended) JSON and parameters:
  FindCount('{name?:?,age:{$gt:?}}',[],['John'],21);
  FindCount('{ ord_dt: { $gt: new Date(?) } }',[],[trunc(Now)-7]);
- optional MaxNumberToReturn can specify a limit for the search (e.g. if you do not want an exact count, but only check for a specific limit)
- optional NumberToSkip can specify the number of matching documents to skip before counting

function FindCount(const Query: variant): Int64; overload;

Calculate the number of documents in the collection that match a specific query
- Criteria can specify the query selector as a BSONVariant/TDocVariant
function FindDoc(const Criteria: Variant; NumberToReturn: integer=1; NumberToSkip: Integer=0; Flags: TMongoQueryFlags=[]): variant; overload;

Select documents in a collection and returns a dvArray TDocVariant instance containing the selected documents
- Criteria can be null (to retrieve all documents) or a TDocVariant / TBSONVariant query selector:
  FindDoc(BSONVariant(’{name:"John",age:{$gt:21}}’),null);
  FindDoc(BSONVariant(’{name?:age:{age:{$gt:??}}’,[],’John’,21)),null);
  see http://docs.mongodb.org/manual/reference/operator for reference
- Projection can be null (to retrieve all fields) or a CSV string to set field names to retrieve, or a TDocVariant or TBSONVariant - e.g.:
  FindDoc(BSONVariant([’name’,’John’]),null);
  FindDoc(BSONVariant([’name’,’John’]),’_id,name’);
  FindDoc(BSONVariant([’name’,’John’]),BSONVariantFieldSelector(’name,_id’));
- NumberToReturn can be left to its default maxInt value to return all matching documents, or specify a limit (e.g. 1 for one document - in this case, the returned instance won't be a dvArray kind of TDocVariant, but either null or the single returned document)
  - if the query does not have any matching record, it will return null

function FindDoc(Criteria: PUTF8Char; const Params: array of const; NumberToReturn: integer=maxInt; NumberToSkip: Integer=0; Flags: TMongoQueryFlags=[]): variant; overload;

Select documents in a collection and returns a dvArray TDocVariant instance containing the selected documents
- Criteria can specify the query selector as (extended) JSON and parameters:
  FindDoc(’{name:"John",age:{$gt:21}}’,[]);
  FindDoc(’{name?:age:{age:{$gt:??}}’,[],’John’,21]);
  see http://docs.mongodb.org/manual/reference/operator for reference
- this overloaded method will use a null Projection, i.e. will retrieve all fields
- NumberToReturn can be left to its default maxInt value to return all matching documents, or specify a limit (e.g. 1 for one document - in this case, the returned instance won't be a dvArray kind of TDocVariant, but either null or the single returned document)
  - if the query does not have any matching record, it will return null

function FindDocs(Criteria: PUTF8Char; const Params: array of const; const Projection: variant; NumberToReturn: integer=maxInt; NumberToSkip: Integer=0; Flags: TMongoQueryFlags=[]): TVariantDynArray; overload;

Select documents in a collection and returns a dynamic array of TDocVariant instance containing the selected documents
- could be used to fill a VCL grid using a TDocVariantArrayDataSet as defined in
  SynVirtualDataSet.pas:
  ds1.DataSet := ToDataSet(self,FindDocs(’{name?:age:{age:{$gt:??}}’,[],’John’,21),null));
- Projection can be null (to retrieve all fields) or a CSV string to set field names to retrieve, or a TDocVariant or TBSONVariant with projection operators
function FindJSON(Criteria: PUTF8Char; const Params: array of const; NumberToReturn: integer=maxInt; NumberToSkip: Integer=0; Flags: TMongoQueryFlags=[]; Mode: TMongoJSONMode=modMongoStrict): RawUTF8; overload;

Select documents in a collection and returns a JSON array of documents containing the selected documents
- Criteria can specify the query selector as (extended) JSON and parameters:
  FindJSON('{name:"John",age:{$gt:21}}',[[],]);
  FindJSON('{name:?},age:{$gt:?}',[null],[John,21]);

see http://docs.mongodb.org/manual/reference/operator for reference
- NumberToReturn can be left to its default maxInt value to return all matching documents as a '[]' JSON array, or specify a limit (e.g. 1 for one document - in this case, the returned instance won't be a '[]' JSON array, but either 'null' or a single '{..}' JSON object)

function FindJSON(const Criteria, Projection: Variant; NumberToReturn: integer=maxInt; NumberToSkip: Integer=0; Flags: TMongoQueryFlags=[]; Mode: TMongoJSONMode=modMongoStrict): RawUTF8; overload;

Select documents in a collection and returns a JSON array of documents containing the selected documents
- Criteria can be null (to retrieve all documents) or a TDocVariant / TBSONVariant query selector:
  FindJSON(BSONVariant('name':"John",age:{$gt:21}),null);
  FindJSON(BSONVariant('name?:',age:{$gt:?}'),[null],[John,21]),null);

see http://docs.mongodb.org/manual/reference/operator for reference
- Projection can be null (to retrieve all fields) or a CSV string to set the field names to retrieve, or a TDocVariant or TBSONVariant - e.g.:
  FindJSON(BSONVariant('name',John'),null);
  FindJSON(BSONVariant('name:',John'),_id');
  FindJSON(BSONVariant('name',John'),BTNOSNObjectIdSelector('name',_id'));

- NumberToReturn can be left to its default maxInt value to return all matching documents as a '[]' JSON array, or specify a limit (e.g. 1 for one document - in this case, the returned instance won't be a '[]' JSON array, but either 'null' or a single '{..}' JSON object)
- this method is very optimized and will convert the BSON binary content directly into JSON, in either modMongoStrict or modMongoShell layout (modNoMongo will do the same as modMongoStrict)

function FindJSON(Criteria: PUTF8Char; const CriteriaParams: array of const; const Projection: variant; NumberToReturn: integer=maxInt; NumberToSkip: Integer=0; Flags: TMongoQueryFlags=[]; Mode: TMongoJSONMode=modMongoStrict): RawUTF8; overload;

Select documents in a collection and returns a JSON array of documents containing the selected documents
- Criteria and Projection can specify the query selector as (extended) JSON and parameters

function FindOne(const _id: variant): variant; overload;

Find an existing document in a collection, by its _id field
- _id will identify the unique document to be retrieved
- returns null, or a TDocVariant instance

function FindOne(const _id: TBSONObjectId): variant; overload;

Find an existing document in a collection, by its _id field
- _id will identify the unique document to be retrieved
- returns null, or a TDocVariant instance
function FindOne(const NameValuePairs: array of const; ReturnNewObjectIfNotFound: boolean=false): variant; overload;

*Find an existing document in a collection, by a custom Criteria value*
- Criteria object, specified as name/value pairs, will identify the unique document to be retrieved 
- returns the found TDocVariant instance
- if the Criteria has no match, return either null or a new object with default values as NameValuePairs if ReturnNewObjectIfNotFound is true

function IsEmpty: boolean;

*Returns TRUE if the collection has no document, FALSE otherwise*
- is much faster than Count, especially for huge collections

function Save(var Document: variant; DocumentObjectID: PBSONObjectID=nil): boolean; overload;

*Updates an existing document or inserts a new document, depending on its document parameter*
- this document should be a TDocVariant (i.e. created via _JsonFast() or _JsonFastFmt()) since we need to check for the _id field, other types will be converted to a TDocVariant instance (via its JSON representation) so it is pointless to use BSONVariant() here
- if the document does not contain an _id field, then the Save() method performs an insert; during the operation, the client will add to the Document variant the _id field and assign it a unique ObjectId - and the method returns FALSE
- if the document contains an _id field, then the save() method performs an upsert, querying the collection on the _id field: if a document does not exist with the specified _id value, the save() method performs an insert; if a document exists with the specified _id value, the save() method performs an update that replaces ALL fields in the existing document with the fields from the document - and the method returns TRUE
- you can optionally retrieve the _id value with the DocumentObjectID pointer

procedure EnsureIndex(const Keys: array of RawUTF8; Ascending: boolean=true; Unique: boolean=false); overload;

*Creates an index on the specified field(s) if the index does not already exist*
- Keys are the corresponding field names
- you can write e.g. to create an ascending index on a given field:
  ```
  book.EnsureIndex(['orderDate']);
  ```

procedure EnsureIndex(const Keys, Options: variant); overload;

*Creates an index on the specified field(s) if the index does not already exist*
- Keys and Options parameters should be TDocVariant (e.g. created via _JsonFast() or _JsonFastFmt()) - and not BSONVariant values
- for ascending/descending indexes, Keys is a document that contains pairs with the name of the field or fields to index and order of the index: value of 1 specifies ascending and of -1 specifies descending
- options is a non-mandatory document that controls the creation of the index -
  ```
  - you can write e.g.
    ```
    book.EnsureIndex(_JsonFast('{ orderDate: 1 }'),null)
    book.EnsureIndex(_ObjFast(['orderDate',1]),null)
    ```
### Synpse.mORMot Framework

#### Software Architect

**Date:** September 16, 2020

---

**procedure** FindDocs(Criteria: PUTF8Char; **const** Params: array of **const**; **var** result: TVariantDynArray; **const** Projection: variant; NumberToReturn: integer=maxInt; NumberToSkip: Integer=0; **Flags**: TMongoQueryFlags=[]); overload;

*Select documents in a collection and returns a dynamic array of TDocVariant instance containing the selected documents*

- you can e.g. fill a res:TVariantDynArray with the following query:
  ```pascal
  FindDocs('{name:?, age:{$gt:?}}', ['John', 21], res, null);
  ```

- Projection can be null (to retrieve all fields) or a CSV string to set field names to retrieve, or a TDocVariant or TBSONVariant with projection operators

**procedure** FindDocs(**var** result: TVariantDynArray; **const** Projection: variant; NumberToReturn: integer=maxInt; NumberToSkip: Integer=0; **Flags**: TMongoQueryFlags=[]); overload;

*Returns a dynamic array of TDocVariant instance containing all documents of a collection*

- Projection can be null (to retrieve all fields) or a CSV string to set field names to retrieve, or a TDocVariant or TBSONVariant with projection operators

**procedure** Insert(**const** Documents: TBSONDocument; **Flags**: TMongoInsertFlags=[]; **NoAcknowledge**: boolean=false); overload;

*Insert one or more documents in the collection*

- Documents is the low-level concatenation of BSON documents, created e.g. with a TBSONWriter stream
- by default, it will follow Client.WriteConcern pattern - but you can set NoAcknowledge = TRUE to avoid calling the getLastError command and increase the execution speed, at the expense of a unsafe process

**procedure** Insert(**const** Documents: array of variant; **Flags**: TMongoInsertFlags=[]; **NoAcknowledge**: boolean=false); overload;

*Insert one or more documents in the collection*

- Documents is an array of TDocVariant (i.e. created via _JsonFast() or _JsonFastFmt()) - or of TBSONVariant (created via BSONVariant())
- by default, it will follow Client.WriteConcern pattern - but you can set NoAcknowledge = TRUE to avoid calling the getLastError command and increase the execution speed, at the expense of a unsafe process

**procedure** Insert(**const** Document: RawUTF8; **const** Params: array of **const**; DocumentObjectID: PBSONObjectID=nil); overload;

*Insert one document, supplied as (extended) JSON and parameters, in the collection*

- supplied JSON could be either strict or in MongoDB Shell syntax:
  ```pascal
  products.insert('{ _id: ?, item: ?, qty: ? }',[1,'card',15]);
  // here _id is forced on the client side
  products.insert('{ item: ?, qty: ? }',[1,'card',15]);
  // here the _id will be created on the client side as an ObjectID
  ```

- you can retrieve the associated ObjectID, as such:
  ```pascal
  var oid: TBSONObjectID;
  ...  
  products.insert('{ item: ?, qty: ? }',[1,'card',15],@oid);
  writeln(oid.ToText);
  ```
procedure InsertJSON(const JSONDocuments: array of PUTF8Char; Flags: TMongoInsertFlags=[]; NoAcknowledge: boolean=false);

*Insert one or more documents in the collection*
- JSONDocuments is an array of JSON objects
- by default, it will follow Client.WriteConcern pattern - but you can set NoAcknowledge = TRUE to avoid calling the getLastError command and increase the execution speed, at the expense of a unsafe process

procedure Remove(const Query: variant; Flags: TMongoDeleteFlags=[]); overload;

*Delete an existing document or several documents in a collection*
- Query parameter should be TDocVariant (i.e. created via _JsonFast() or _JsonFastFmt()) or TBSONVariant (created via BSONVariant())
- Query is the selection criteria for the deletion; use the same query selectors as used in the Find() method
- to limit the deletion to just one document, set Flags to [mdfSingleRemove]
- to delete all documents matching the deletion criteria, leave it to []

procedure RemoveFmt(Query: PUTF8Char; const QueryParams: array of const; Flags: TMongoDeleteFlags=[]);

*Delete an existing document or several documents in a collection*
- Query parameter can be specified as JSON objects with parameters
- Query is the selection criteria for the deletion; use the same query selectors as used in the Find() method
- to limit the deletion to just one document, set Flags to [mdfSingleRemove]
- to delete all documents matching the deletion criteria, leave it to []

procedure RemoveOne(const _id: TBSONObjectID); overload;

*Delete an existing document in a collection, by its _id field*
- _id will identify the unique document to be deleted

procedure RemoveOne(const _id: variant); overload;

*Delete an existing document in a collection, by its _id field*
- _id will identify the unique document to be deleted

procedure Save(const Document: RawUTF8; const Params: array of const; DocumentObjectID: PBSONObjectID=nil); overload;

*Updates an existing document or inserts a new document, depending on its document parameter, supplied as (extended) JSON and parameters*
- supplied JSON could be either strict or in MongoDB Shell syntax:
- will perform either an insert or an update, depending of the presence of the _id field, as overloaded Save(const Document: variant)
procedure Update(const Query, Update: variant; Flags: TMongoUpdateFlags=[]);

Overload:

Modifies an existing document or several documents in a collection
- the method can modify specific fields of existing document or documents or replace an existing
document entirely, depending on the update parameter
- Query and Update parameters should be TDocVariant (i.e. created via _JsonFast() or
_JsonFastFmt()) or TBSONVariant (created via BSONVariant())
- Query is the selection criteria for the update; use the same query selectors as used in the Find() method
- if Update contains a plain document, it will replace any existing data
- if Update contains update operators (like $set), it will update the corresponding fields in the document

procedure Update(Query: PUTF8Char; const QueryParams: array of const; const Update: RawUTF8; const UpdateParams: array of const; Flags: TMongoUpdateFlags=[])); overload;

Modifies an existing document or several documents in a collection
- the method can modify specific fields of existing document or documents or replace an existing
document entirely, depending on the update parameter
- since all content will be transformed into JSON internally, use this method only if the supplied
parameters are simple types: any complex value (e.g. a TDateTime or a BSONVariant binary)
won't be handled as expected - use the overloaded Update() with explicit BSONVariant() values instead
- Query and Update parameters can be specified as JSON objects with parameters
- Query is the selection criteria for the update; use the same query selectors as used in the Find() method
- if Update contains a plain document, it will replace any existing data:
  people.update({name:?},["Andy"],[name?:?,age?:?],["Andy",25],[mufUpsert]);
Warning: to avoid inserting the same document more than once, only use mufUpsert if the
query field is uniquely indexed
- if Update contains update operators (like $set), it will update the corresponding fields in the document:
  book.insert({_id?:?,item?:?,stock?:?},[11,'Divine Comedy',2]);
  book.update({item?:?},{"Divine Comedy"},{$set:{price?:?},$inc:{stock?:?}},[18,5]);
  // the updated document is now:
  { "_id" : 11, "item" : "Divine Comedy", "price" : 18, "stock" : 7 }

procedure UpdateOne(const _id, UpdatedFields: variant);

Modifies some fields of an existing document in a collection
- by default, Update() or Save() will replace the whole document
- this method will expect the identifier to be supplied as a variant - may be via the ObjectId() function
- and will replace the specified fields, i.e. it will execute a $set: with the supplied UpdatedFields
value

property Database: TMongoDatabase read fDatabase;

The associated MongoDB database instance

property FullCollectionName: RawUTF8 read fFullCollectionName;

The full collection name, e.g. 'dbname.collectionname'
property Name: RawUTF8 read fName;  
   The collection name

EMongoConnectionException = class(EMongoException)  
Exception type used for MongoDB process, once connected

constructor Create(const aMsg: string; aConnection: TMongoConnection); reintroduce; overload;
   Initialize the Exception for a given request

constructor CreateUTF8(const Format: RawUTF8; const Args: array of const; 
aConnection: TMongoConnection); reintroduce;
   Initialize the Exception for a given request

property Connection: TMongoConnection read fConnection;  
   The associated connection

EMongoRequestException = class(EMongoConnectionException)  
Exception type used for MongoDB query process

constructor Create(const aMsg: string; aConnection: TMongoConnection; aRequest: TMongoRequest; const aError: TMongoReplyCursor); reintroduce; overload;
   Initialize the Exception for a given request

constructor Create(const aMsg: string; aConnection: TMongoConnection; aRequest: TMongoRequest; aRequest: TMongoRequest=nil); reintroduce; overload;
   Initialize the Exception for a given request

constructor CreateUTF8(const Format: RawUTF8; const Args: array of const; 
aConnection: TMongoConnection; aRequest: TMongoRequest); reintroduce;
   Initialize the Exception for a given request

function CustomLog(WR: TTextWriter; const Context: TSynLogExceptionContext): boolean; override;
   Used to customize the exception log to contain information about the Query  
   - it will log both the failing request and the returned error message

property ErrorDoc: Variant read GetErrorDoc;  
   The associated error reply document, as a TDocVariant instance  
   - will return the first document available in ErrorReply, or the supplied aErrorDoc: TDocVariantData instance

property ErrorReply: TMongoReplyCursor read fError;  
   The associated error reply document
EMongoRequestOSException = class(EMongoRequestException)
Exception type used for MongoDB query process after an Operating System error (e.g. in case of socket error)

constructor Create(const aMsg: string; aConnection: TMongoConnection; aRequest: TMongoRequest=nil); reintroduce;
Initialize the Exception for a given request, including the last error message retrieved from the operating system
- if such an exception is raised, you can use SystemLastError property to retrieve the corresponding Operating System error code

property SystemLastError: cardinal read fSystemLastError;
Contain the associated Operating System last error code
- will specify e.g. the kind of communication/socket error

Types implemented in the SynMongoDB unit

PBSON24 = ^TBSON24;
Points to 24-bit storage, mapped as a 3 bytes buffer

PBSONElementType = ^TBSONElementType;
Points to an element type for BSON internal representation

PBSONObjectID = ^TBSONObjectID;
Points to a BSON ObjectID internal binary representation

PBSONVariantData = ^TBSONVariantData;
Points to memory structure used for some special BSON storage as variant

PDecimal128 = ^TDecimal128;
Points to a 128-bit decimal value

PDecimal128Bits = ^TDecimal128Bits;
Points to a 128-bit decimal binary

PMongoReplyHeader = ^TMongoReplyHeader;
Points to an low-level binary structure mapping the TMongoReply header
- so that you can write e.g. PMongoReplyHeader(aMongoReply)^.RequestID

TBSONDcArrayConversion = ( asBSONVariant, asDocVariantPerValue, asDocVariantPerReference, asDocVariantInternNamesPerValue, asDocVariantInternNamesPerReference );
Define how betDoc/betArray BSON elements will be converted as variants
- by default a TBSONVariant custom type will be returned, containing the raw BSON binary content of the embedded document or array
- asDocVariantPerValue or asDocVariantPerReference could be used to create a tree of TDocVariant custom kind of variant, able to access to its nested properties via late-binding (asDocVariantPerReference being also much faster in some cases - but less safe - than asDocVariantPerValue)
- asDocVariantPerValue will set JSON_OPTIONS[false] settings:
asDocVariantPerReference will set JSON_OPTIONS[true]/JSON_OPTIONS_FAST settings:
- [dvoValueCopiedByReference,dvoReturnNullForUnknownProperty]
- asDocVariantInternNamesPerValue and asDocVariantInternNamesPerReference will include
dvoInternalNames to the TDocVariant.Options

```pascal
TBSONDocument = RawByteString;
```

- Storage of a BSON binary document
- a specific type is defined for consistency with this unit classes
- binary content should follow the "int32 e_list #0" standard layout

```pascal
TBSONDocumentDynArray = array of TBSONDocument;
```

- Dynamic array of BSON binary document storage

```pascal
TBSONElementBinaryType = ( bbtGeneric, bbtFunction, bbtOldBinary, bbtOldUUID, bbtUUID,
bbtMD5, bbtUser );
```

- Sub-types for binary element BSON internal representation

```pascal
TBSONElementType = ( betEOF, betFloat, betString, betDoc, betArray, betBinary,
betDeprecatedUndefined, betObjectID, betBoolean, betDateTime, betNull, betRegEx,
betDeprecatedDbptr, betJS, betDeprecatedSymbol, betJSScope, betInt32, betTimestamp,
betInt64, betDecimal128 );
```

- Element types for BSON internal representation

```pascal
TDecimal128SpecialValue = ( dsvError, dsvValue, dsvNan, dsvZero, dsvPosInf, dsvNegInf,
dsvMin, dsvMax );
```

- Some special 128-bit decimal values
- see TDecimal128.SetSpecial to set the corresponding value
- dsvError is returned by TDecimal128.FromText() on parsing error
- dsvValue indicates that this is not a known "special" value, but some valid decimal number

```pascal
TDecimal128Str = array[0..42] of AnsiChar;
```

- Enough characters to contain any TDecimal128 text representation

```pascal
TMongoClientReplicaSetReadPreference = ( rpPrimary, rpPrimaryPreferred, rpSecondary,
rpSecondaryPreferred );
```

- Define Read Preference Modes to a MongoDB replica set
- Important: All read preference modes except rpPrimary may return stale data because secondaries
replicate operations from the primary with some delay - ensure that your application can tolerate
stale data if you choose to use a non-primary mode
- rpPrimary: Default mode - all operations read from the current replica set primary
- rpPrimaryPreferred: in most situations, operations read from the primary but if it is unavailable,
operations read from secondary members.
- rpSecondary: all operations read from the secondary members of the replica set
- rpSecondaryPreferred: in most situations, operations read from secondary members but if no
secondary members are available, operations read from the primary

```pascal
TMongoClientWriteConcern = ( wcAcknowledged, wcJournalized, wcReplicaAcknowledged,
wcUnacknowledged, wcErrorsIgnored );
```

- Define Write Concern property of a MongoDB connection
- Write concern describes the guarantee that MongoDB provides when reporting on the success of a
write operation. The strength of the write concerns determine the level of guarantee. When inserts,
updates and deletes have a weak write concern, write operations return quickly. In some failure
cases, write operations issued with weak write concerns may not persist. With stronger write
concerns, clients wait after sending a write operation for MongoDB to confirm the write operations. MongoDB provides different levels of write concern to better address the specific needs of applications. Clients may adjust write concern to ensure that the most important operations persist successfully to an entire MongoDB deployment. For other less critical operations, clients can adjust the write concern to ensure faster performance rather than ensure persistence to the entire deployment.

- wcAcknowledged is the default safe mode: the mongod confirms the receipt of the write operation. Acknowledged write concern allows clients to catch network, duplicate key, and other errors.
- with wcJournaled, the mongod acknowledges the write operation only after committing the data to the journal. This write concern ensures that MongoDB can recover the data following a shutdown or power interruption.
- wcReplicaAcknowledged will guarantee that the write operation propagates to at least one member of a replica set
- with wcUnacknowledged, MongoDB does not acknowledge the receipt of write operation. Unacknowledged is similar to errors ignored; however, drivers attempt to receive and handle network errors when possible. The driver’s ability to detect network errors depends on the system's networking configuration.
- with wcErrorsIgnored, MongoDB does not acknowledge write operations. With this level of write concern, the client cannot detect failed write operations. These errors include connection errors and mongod exceptions such as duplicate key exceptions for unique indexes. Although the errors ignored write concern provides fast performance, this performance gain comes at the cost of significant risks for data persistence and durability. WARNING: Do not use wcErrorsIgnored write concern in normal operation.

TMongoConnectionDynArray = array of TMongoConnection;

  Array of TCP connection to a MongoDB Replica Set
  - first item [0] is the Primary member
  - other items [1..] are the Secondary members

TMongoDeleteFlag = ( mdfSingleRemove );

  Define how an opDelete operation will behave
  - if mdfSingleRemove is set, the database will remove only the first matching document in the collection. Otherwise (by default) all matching documents will be removed

TMongoDeleteFlags = set of TMongoDeleteFlag;

  Define how a TMongoRequestDelete message will behave

TMongoInsertFlag = ( mifContinueOnError );

  Define how an opInsert operation will behave
  - if mifContinueOnError is set, the database will not stop processing a bulk insert if one fails (e.g. due to duplicate IDs); this makes bulk insert behave similarly to a series of single inserts, except lastError will be set if any insert fails, not just the last one - if multiple errors occur, only the most recent will be reported by getLastError

TMongoInsertFlags = set of TMongoInsertFlag;

  Define how a TMongoRequestInsert message will behave

TMongoJSONMode = ( modNoMongo, modMongoStrict, modMongoShell );

  How TBSONElement.AddMongoJSON() method and AddMongoJSON() and VariantSaveMongoJSON() functions will render their JSON content
  - modMongoStrict and modNoMongo will follow the JSON RFC specifications
modMongoShell will use a syntax incompatible with JSON RFC, but more common to MongoDB daily use - as 'ObjectID()' or '{ field: /acme.*corp/i }'
modMongoStrict will use the MongoDB Extended JSON syntax
modNoMongo will serialize dates as ISO-8601 strings, ObjectID as hexadecimal string and other MongoDB special objects in WrBase64() format
- see http://docs.mongodb.org/manual/reference/mongodb-extended-json

```pascal
TMongoOperation = ( opReply, opMsgOld, opUpdate, opInsert, opQuery, opGetMore, opDelete, opKillCursors, opMsg );
```

The available MongoDB driver Request Opcodes
- opReply: database reply to a client request - ResponseTo shall be set
- opMsgOld: generic msg command followed by a string (deprecated)
- opUpdate: update document
- opInsert: insert new document
- opQuery: query a collection
- opGetMore: get more data from a previous query
- opDelete: delete documents
- opKillCursors: notify database client is done with a cursor
- opMsg: new OP_MSG layout introduced in MongoDB 3.6

```pascal
TMongoQueryFlag = ( mqfTailableCursor, mqfSlaveOk, mqfOplogReplay, mqfNoCursorTimeout, mqfAwaitData, mqfExhaust, mqfPartial );
```

Define how an opQuery operation will behave
- if mqfTailableCursor is set, cursor is not closed when the last data is retrieved
- if mqfSlaveOk is set, it will allow query of replica slave; normally this returns an error except for namespace "local"
- mqfOplogReplay is internal replication use only - driver should not set
- if mqfNoCursorTimeout is set, the server normally does not times out idle cursors after an inactivity period (10 minutes) to prevent excess memory use
- if mqfAwaitData is to use with TailableCursor. If we are at the end of the data, block for a while rather than returning no data. After a timeout period, we do return as normal
- if mqfExhaust is set, stream the data down full blast in multiple "more" packages, on the assumption that the client will fully read all data queried
- if mqfPartial is set, it will get partial results from a mongos if some shards are down (instead of throwing an error)

```pascal
TMongoQueryFlags = set of TMongoQueryFlag;
```

Define how a TMongoRequestQuery message will behave

```pascal
TMongoReply = RawByteString;
```

Used to store the binary raw data a database response to a TMongoRequestQuery / TMongoRequestGetMore client message

```pascal
TMongoReplyCursorFlag = ( mrfCursorNotFound, mrfQueryFailure, mrfShardConfigStale, mrfAwaitCapable );
```

Define an opReply message execution content
- mrfCursorNotFound will be set when getMore is called but the cursor id is not valid at the server; returned with zero results
- mrfQueryFailure is set when the query failed - results consist of one document containing an "$err" field describing the failure
- mrfShardConfigStale should not be used by client, just by Mongos
- mrfAwaitCapable is set when the server supports the AwaitData Query option (always set since
Mongod version 1.6)

`TMongoReplyCursorFlags = set of TMongoReplyCursorFlag;`

*Define a `TMongoReplyCursor` message execution content*

`TMongoUpdateFlag = ( mufUpsert, mufMultiUpdate );`

*Define how an `opUpdate` operation will behave*

- if `mufUpsert` is set, the database will insert the supplied object into the collection if no matching document is found
- if `mufMultiUpdate` is set, the database will update all matching objects in the collection; otherwise (by default) only updates first matching doc

`TMongoUpdateFlags = set of TMongoUpdateFlag;`

*Define how an `TMongoRequestUpdate` message will behave*

`TOnMongoConnectionReply = procedure(Request: TMongoRequest; const Reply: TMongoReplyCursor; var Opaque) of object;`

*Event callback signature for iterative process of `TMongoConnection`*

**Constants implemented in the `SynMongoDB` unit**

`betMaxKey = $7f;`

*Fake BSON element type which compares higher than all other possible values*

- element type sounds to be stored as shortint, so here `betInt64=$12<$7f`
- defined as an integer to circumvent a compilation issue with FPC trunk

`betMinKey = $ff;`

*Fake BSON element type which compares lower than all other possible values*

- element type sounds to be stored as shortint, so here $ff=-1<0=betEOF`
- defined as an integer to circumvent a compilation issue with FPC trunk

`BSON_DECIMAL128_HI_CURRNEG = $b038000000000000;`

*4 fixed decimals*

`BSON_DECIMAL128_HI_INT64NEG = $b040000000000000;`

*0 fixed decimals*

`BSON_DECIMAL128_HI_NAN = $7c00000000000000;`

`DsvError, dsvValue, dsvNaN, dsvZero, dsvPosInf, dsvNegInf, dsvMin, dsvMax`

`BSON_ELEMENTVARIANTMANAGED = [betBinary, betDoc, betArray, betRegEx, betDeprecatedDbptr, betTimestamp, betJSScope, betJS, betDeprecatedSymbol, betDecimal128];`

*Kind of elements which will store a RawByteString/RawUTF8 content within its TBSONVariant kind*

- i.e. TBSONVariantData.VBlob/VText field is to be managed

`BSON_JSON_BINARY: array[boolean,boolean] of string[15] = ( ('"$binary":"','"$type":"'),('BinData(','"',));`

*Special JSON patterns which will be used to format a betBinary item*

- *false,* is for strict JSON, *true,* for MongoDB Extended JSON

`BSON_JSON_DATE: array[TMongoJSONMode,boolean] of string[15] = ( ('"','"'),('{"$date":"','""'),('ISODate("','"'));`

*Special JSON patterns which will be used to format a betDateTime item*

- *false,* is to be written before the date value, *true,* after
BSON_JSON_DBREF: array[boolean, 0..2] of string[15] = (
  ('"$ref":"","$id":""'), ('DBRef("","")'));

Special JSON string content which will be used to store a deprecated Dbptr
- *[false,*] is for strict JSON, *[true,*] for MongoDB Extended JSON
- (not used by now for this deprecated content)

BSON_JSON_DECIMAL: array[boolean, TMongoJSONMode] of string[23] = (  
  ('"",'"$numberDecimal":"",'NumberDecimal("",""",")")');

Special JSON patterns which will be used to format a deprecated DECIMAL128 item
- *[false,*] is to be written before the decimal value, *[true,*] after

BSON_JSON_MAXKEY: array[boolean] of string[15] = ( '"$maxKey":1', 'MaxKey' );

Special JSON string content which will be used to store a maxKey item
- *[false,*] is for strict JSON, *[true,*] for MongoDB Extended JSON

BSON_JSON_MINKEY: array[boolean] of string[15] = ( '"$minKey":1', 'MinKey' );

Special JSON string content which will be used to store a minKey item
- *[false,*] is for strict JSON, *[true,*] for MongoDB Extended JSON

  ('"",'"$oid":"",'ObjectId("",")")');

Special JSON patterns which will be used to format a ObjectId item
- *[false,*] is to be written before the hexadecimal ID, *[true,*] after

BSON_JSON_REGEX: array[0..2] of string[15] = ( '"$regex":"",'"$options":""');

Special JSON string content which will be used to store a RegEx

BSON_JSON_UNDEFINED: array[boolean] of string[23] =  
  ( '"$undefined":true', 'undefined' );

Special JSON string content which will be used to store a deprecated Undefined item
- *[false,*] is for strict JSON, *[true,*] for MongoDB Extended JSON

BSON_MAXDOCUMENTSIZE = 16*1024*1024;

By definition, maximum MongoDB document size is 16 MB

DECIMAL128_SPECIAL_TEXT: array[TDecimal128SpecialValue] of RawUTF8 = ( '','','NaN', '0', 'Infinity', '-Infinity', '-.9.99999999999999999999999999999999E+6144', '9.99999999999999999999999999999999E+6144' );

The textual representation of the TDecimal128 special values

MONGODB_DEFAULTPORT = 27017;

MongoDB server default IP port

Functions or procedures implemented in the SynMongoDB unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddMongoJSON</td>
<td>Convert any kind of BSON/JSON element, encoded as variant, into JSON</td>
<td>1443</td>
</tr>
<tr>
<td>BSON</td>
<td>Store some object content, supplied as (extended) JSON and parameters, into BSON encoded binary</td>
<td>1445</td>
</tr>
<tr>
<td>BSON</td>
<td>Store some TDocVariant custom variant content into BSON encoded binary</td>
<td>1445</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>BSON</td>
<td>Store some object content, supplied as (extended) JSON, into BSON encoded binary</td>
<td>1445</td>
</tr>
<tr>
<td>BSON</td>
<td>Store some object content into BSON encoded binary</td>
<td>1445</td>
</tr>
<tr>
<td>BSONDocumentToDoc</td>
<td>Convert a TBSONDocument into a TDocVariant variant instance</td>
<td>1445</td>
</tr>
<tr>
<td>BSONDocumentToJson</td>
<td>Convert a TBSONDocument into its JSON representation</td>
<td>1445</td>
</tr>
<tr>
<td>BSONFieldSelector</td>
<td>Create a fields selector BSON document from a CSV field names list</td>
<td>1445</td>
</tr>
<tr>
<td>BSONFieldSelector</td>
<td>Create a fields selector BSON document from a field names list</td>
<td>1445</td>
</tr>
<tr>
<td>BSONFromInt64s</td>
<td>Store an array of 64 bit integer into BSON encoded binary</td>
<td>1445</td>
</tr>
<tr>
<td>BSONFromIntegers</td>
<td>Store an array of integer into BSON encoded binary</td>
<td>1445</td>
</tr>
<tr>
<td>BSONListToJson</td>
<td>Convert a BSON list of elements into its JSON representation</td>
<td>1445</td>
</tr>
<tr>
<td>BSONObjectId</td>
<td>Convert a TBSONVariant Object ID custom variant into a TBSONObjectID</td>
<td>1446</td>
</tr>
<tr>
<td>BSONParseLength</td>
<td>Parse the header of a BSON encoded binary buffer, and return its length</td>
<td>1446</td>
</tr>
<tr>
<td>BSONParseNextElement</td>
<td>Parse the next element in supplied BSON encoded binary buffer list</td>
<td>1446</td>
</tr>
<tr>
<td>BSONPerIndexElement</td>
<td>Search for a property by number in a supplied BSON encoded binary buffer</td>
<td>1446</td>
</tr>
<tr>
<td>BSONToDoc</td>
<td>Convert a BSON document into a TDocVariant variant instance</td>
<td>1446</td>
</tr>
<tr>
<td>BSONToJson</td>
<td>Convert a BSON document into its JSON representation</td>
<td>1447</td>
</tr>
<tr>
<td>BSONVariant</td>
<td>Convert a TDocVariant variant into a TBSONVariant betDoc type instance</td>
<td>1447</td>
</tr>
<tr>
<td>BSONVariant</td>
<td>Store some object content into a TBSONVariant betDoc type instance</td>
<td>1447</td>
</tr>
<tr>
<td>BSONVariant</td>
<td>Store some object content, supplied as (extended) JSON and parameters, into a TBSONVariant betDoc type instance</td>
<td>1447</td>
</tr>
<tr>
<td>BSONVariant</td>
<td>Store some object content, supplied as (extended) JSON, into a TBSONVariant betDoc type instance</td>
<td>1447</td>
</tr>
<tr>
<td>BSONVariantFieldSelector</td>
<td>Create a fields selector BSON document from a CSV field names list</td>
<td>1447</td>
</tr>
<tr>
<td>BSONVariantFieldSelector</td>
<td>Create a fields selector BSON document from a field names list</td>
<td>1447</td>
</tr>
<tr>
<td>BSONVariantFromInt64s</td>
<td>Store an array of 64 bit integer into a TBSONVariant betArray type instance</td>
<td>1448</td>
</tr>
</tbody>
</table>
### Functions or procedures

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BSONVariantFromIntegers</strong></td>
<td>Store an array of integer into a TBSONVariant betArray type instance</td>
<td>1448</td>
</tr>
<tr>
<td><strong>JavaScript</strong></td>
<td>Create a TBSONVariant JavaScript custom variant type from a supplied code</td>
<td>1448</td>
</tr>
<tr>
<td><strong>JavaScript</strong></td>
<td>Create a TBSONVariant JavaScript and associated scope custom variant type from a supplied code and document</td>
<td>1448</td>
</tr>
<tr>
<td><strong>JSONBufferToBSONArray</strong></td>
<td>Store one JSON array into an array of BSON binary</td>
<td>1448</td>
</tr>
<tr>
<td><strong>JSONBufferToBSONDocument</strong></td>
<td>Store some object content, supplied as (extended) JSON into BSON binary</td>
<td>1448</td>
</tr>
<tr>
<td><strong>NumberDecimal</strong></td>
<td>Create a TBSONVariant Decimal128 from some text corresponding to a floating-point number</td>
<td>1448</td>
</tr>
<tr>
<td><strong>NumberDecimal</strong></td>
<td>Create a TBSONVariant Decimal128 from a currency fixed decimal</td>
<td>1448</td>
</tr>
<tr>
<td><strong>ObjectID</strong></td>
<td>Create a TBSONVariant Object ID custom variant type from a supplied text</td>
<td>1449</td>
</tr>
<tr>
<td><strong>ObjectID</strong></td>
<td>Create a TBSONVariant custom variant type containing a BSON Object ID</td>
<td>1449</td>
</tr>
<tr>
<td><strong>ToText</strong></td>
<td>Ready-to-be displayed text of a TBSONElementTypeName value</td>
<td>1449</td>
</tr>
<tr>
<td><strong>ToText</strong></td>
<td>Ready-to-be displayed text of a TDecimal128SpecialValue</td>
<td>1449</td>
</tr>
<tr>
<td><strong>ToText</strong></td>
<td>Ready-to-be displayed text of a TMongoClientReplicaSetReadPreference item</td>
<td>1449</td>
</tr>
<tr>
<td><strong>ToText</strong></td>
<td>Ready-to-be displayed text of a TMongoOperation item</td>
<td>1449</td>
</tr>
<tr>
<td><strong>ToText</strong></td>
<td>Ready-to-be displayed text of a TMongoClientWriteConcern item</td>
<td>1449</td>
</tr>
<tr>
<td><strong>VariantSaveMongoJSON</strong></td>
<td>Convert any kind of BSON/JSON element, encoded as variant, into JSON</td>
<td>1449</td>
</tr>
</tbody>
</table>

---

```pascal
procedure AddMongoJSON(const Value: variant; W: TTextWriter; Mode: TMongoJSONMode=modMongoStrict); overload;

Convert any kind of BSON/JSON element, encoded as variant, into JSON
- this function will use by default the MongoDB Extended JSON syntax for specific MongoDB objects but you may use modMongoShell if needed
```
function BSON(const JSON: RawUTF8; kind: PBSONElementType=nil): TBSONDocument; overload;

*Store some object content, supplied as (extended) JSON, into BSON encoded binary*
- in addition to the JSON RFC specification strict mode, this method will handle some BSON-like extensions, e.g. unquoted field names:
  
  ```
  BSON('{id:10,doc:{name:"John",birthyear:1972}}');
  ```
- MongoDB Shell syntax will also be recognized to create TBSONVariant, like
  ```
  new Date()   ObjectId()   MinKey   MaxKey  /<jRegex>/<jOptions>
  ```
  see @http://docs.mongodb.org/manual/reference/mongodb-extended-json

  ```
  BSON('{id:new ObjectId(),doc:{name:"John",date:ISODate()}}');
  ```
  ```
  BSON('{name:"John",field:/acme.*corp/i}');
  ```
- will create the BSON binary without any temporary TDocVariant storage, by calling
  BSONToJSONDocument() on a temporary copy of the supplied JSON

function BSON(const NameValuePairs: array of const): TBSONDocument; overload;

*Store some object content into BSON encoded binary*
- object will be initialized with data supplied two by two, as Name,Value pairs, e.g.:
  ```
  aBson := BSON(['name','John','year',1972]);
  ```
- you can define nested arrays or objects as TDocVariant, e.g.
  ```
  aBSON := BSON([''bsonDat'','Arr([''awesome'',5.05,1986]])];
  ```
- or you can specify nested arrays or objects with ['..'] or '...
  ```
  aBSON := BSON([''BSON'',''['',''id'',123]]);
  ```
- will create the BSON binary without any temporary TDocVariant storage

function BSON(const Format: RawUTF8; const Args,Params: array of const; kind: PBSONElementType=nil): TBSONDocument; overload;

*Store some object content, supplied as (extended) JSON and parameters, into BSON encoded binary*
- in addition to the JSON RFC specification strict mode, this method will handle some BSON-like extensions, e.g. unquoted field names
- MongoDB Shell syntax will also be recognized to create TBSONVariant, like
  ```
  new Date()   ObjectId()   MinKey   MaxKey  /<jRegex>/<jOptions>
  ```
  see @http://docs.mongodb.org/manual/reference/mongodb-extended-json
- typical use could be:
  ```
  BSON('%{$in:[?,?]}',['type',['food','snack']]);
  BSON('type:{$in:[]}'],[,['food','snack']]])];
  BSON('%[?:?,?,?]',['BSON'],[Arr(['awesome',5.05,1986])])
  BSON('%[?],?[\BSON],Arr(['awesome',5.05,1986])])
  BSON('name:field:/%/i',['acme.*corp'],['John'])
  BSON('id:new ObjectIdId(),doc:name:',date:ISODate())']},[],'John',NowUTC]);
  ```
- will create the BSON binary without any temporary TDocVariant storage, by calling
  BSONToJSONDocument() on the generated JSON content
- since all content will be transformed into JSON internally, use this method only if the supplied parameters are simple types, and identified explicitly via BSON-like extensions: any complex value (e.g. a TDateTime or a BSONVariant binary) won't be handled as expected - use the overloaded BSON() with explicit BSONVariant() name/value pairs instead


function BSON(const doc: TDocVariantData): TBSONDocument; overload;

- Store some TDocVariant custom variant content into BSON encoded binary
- will write either a BSON object or array, depending on the internal layout of this TDocVariantData instance (i.e. Kind property value)
- if supplied variant is not a TDocVariant, raise an EBSONException

function BSONDocumentToDoc(const BSON: TBSONDocument; Option: TBSONDocArrayConversion=asDocVariantPerReference): variant;

- Convert a TBSONDocument into a TDocVariant variant instance
- BSON should be valid BSON document (length will be checked against expected "int32 e_list #0" binary layout)
- by definition, asBSONVariant is not allowed as Option value

function BSONDocumentToJSON(const BSON: TBSONDocument; Mode: TMongoJSONMode=modMongoStrict): RawUTF8;

- Convert a TBSONDocument into its JSON representation
- BSON should be valid BSON document (length will be checked against expected "int32 e_list #0" binary layout)
- this function will use by default the MongoDB Extended JSON syntax for specific MongoDB objects but you may use modMongoShell if needed

function BSONFieldSelector(const FieldNames: array of RawUTF8): TBSONDocument; overload;

- Create a fields selector BSON document from a field names list
- can be used via a TBSONVariant instance for the projection parameter of a TMongoRequestQuery, e.g.
  BSONToJSON(BSONFieldSelector(['a', 'b', 'c'])) = '{"a":1,"b":1,"c":1}'

function BSONFieldSelector(const FieldNamesCSV: RawUTF8): TBSONDocument; overload;

- Create a fields selector BSON document from a CSV field names list
- can be used via a TBSONVariant instance for the projection parameter of a TMongoRequestQuery, e.g.
  BSONToJSON(BSONFieldSelector('a,b,c')) = '{"a":1,"b":1,"c":1}'

function BSONFromInt64s(const Integers: array of Int64): TBSONDocument;

- Store an array of 64 bit integer into BSON encoded binary
- object will be initialized with data supplied e.g. as a TIntegerDynArray

function BSONFromIntegers(const Integers: array of integer): TBSONDocument;

- Store an array of integer into BSON encoded binary
- object will be initialized with data supplied e.g. as a TIntegerDynArray

procedure BSONListToJSON(BSONList: PByte; Kind: TBSONElementType; W: TTextWriter; Mode: TMongoJSONMode=modMongoStrict);

- Convert a BSON list of elements into its JSON representation
- BSON should point to the "e_list" of the "int32 e_list #0" BSON document, i.e. the item data as expected by TBSONElement.FromNext()
- this function will use by default the MongoDB Extended JSON syntax for specific MongoDB objects but you may use modMongoShell if needed
function BSONObjectID(const aObjectID: variant): TBSONObjectID;
  Convert a TBSONVariant Object ID custom variant into a TBSONObjectID
  - raise an exception if the supplied variant is not a TBSONVariant Object ID

function BSONParseLength(var BSON: PByte; ExpectedBSONLen: integer=0): integer;
  Parse the header of a BSON encoded binary buffer, and return its length
  - BSON should point to a "int32 e_list #0" BSON document (like TBSONDocument)
  - if ExpectedBSONLen is set, this function will check that the supplied BSON content "int32" length
    matches the supplied value, and raise an EBSONException if this comparison fails
  - as an alternative, consider using TBSONIterator, which wrap both a PByte and a TBSONElement
    into one convenient item

function BSONParseNextElement(var BSON: PByte; var name: RawUTF8; var element: variant; DocArrayConversion: TBSONDocArrayConversion=asBSONVariant): boolean;
  Parse the next element in supplied BSON encoded binary buffer list
  - BSON should point to the "e_list" of the "int32 e_list #0" BSON document
  - will decode the supplied binary buffer as a variant, then it will let BSON point to the next
    element, and return TRUE
  - returns FALSE when you reached betEOF, so that you can use it in a loop:
    var bson: PByte;
    name: RawUTF8;
    value: variant;
    ...
    BSONParseLength(bson);
    while BSONParseNextElement(bson,name,value) do
      writeln(name,' :' ,value);
  - by default, it will return TBSONVariant custom variants for documents or arrays - but if
    storeDocArrayAsDocVariant is set, it will return a TDocVariant custom kind of variant, able to
    access to its nested properties via late-binding
  - if you want to parse a BSON list as fast as possible, you should better use
    TBSONElement.FromNext() which avoid any memory allocation (the SAX way) - in fact, this
    function is just a wrapper around TBSONElement.FromNext + ToVariant
  - as an alternative, consider using TBSONIterator, which wrap both a PByte and a TBSONElement
    into one convenient item

function BSONPerIndexElement(BSON: PByte; index: integer; var item: TBSONElement): boolean;
  Search for a property by number in a a supplied BSON encoded binary buffer
  - BSON should point to a "int32 e_list #0" BSON document (like TBSONDocument)
  - returns FALSE if the list has too few elements (starting at index 0)
  - otherwise, returns TRUE then let item point to the corresponding element

procedure BSONToDoc(BSON: PByte; var Result: Variant; ExpectedBSONLen: integer=0; Option: TBSONDocArrayConversion=asBSONVariantPerReference);
  Convert a BSON document into a TDocVariant variant instance
  - BSON should point to a "int32 e_list #0" BSON document
  - if ExpectedBSONLen is set, this function will check that the supplied BSON content "int32" length
    matches the supplied value
  - by definition, asBSONVariant is not allowed as Option value
**function** BSONToJSON(BSON: PByte; Kind: TBSONElementType; ExpectedBSONLen: integer=0; Mode: TMongoJSONMode=modMongoStrict): RawUTF8;

*Convert a BSON document into its JSON representation*
- BSON should point to a "int32 e_list #0" BSON document
- Kind should be either betDoc or betArray
- if ExpectedBSONLen is set, this function will check that the supplied BSON content "int32" length matches the supplied value
- this function will use by default the MongoDB Extended JSON syntax for specific MongoDB objects but you may use modMongoShell if needed

**procedure** BSONVariant(JSON: PUTF8Char; var result: variant); overload;

*Store some object content, supplied as (extended) JSON, into a TBSONVariant betDoc type instance*
- in addition to the JSON RFC specification strict mode, this method will handle some BSON-like extensions, as with the overloaded BSON() function
- warning: this overloaded method will modify the supplied JSON buffer in-place: you can use the overloaded BSONVariant(const JSON: RawUTF8) function instead if you do not want to modify the input buffer content

**function** BSONVariant(const JSON: RawUTF8): variant; overload;

*Store some object content, supplied as (extended) JSON, into a TBSONVariant betDoc type instance*
- in addition to the JSON RFC specification strict mode, this method will handle some BSON-like extensions, as with the overloaded BSON() function

**function** BSONVariant(const Format: RawUTF8; const Args,Params: array of const): variant; overload;

*Store some object content, supplied as (extended) JSON and parameters, into a TBSONVariant betDoc type instance*
- in addition to the JSON RFC specification strict mode, this method will handle some BSON-like extensions, as with the overloaded BSON() function

**function** BSONVariant(doc: TDocVariantData): variant; overload;

*Convert a TDocVariant variant into a TBSONVariant betDoc type instance*

**function** BSONVariant(const NameValuePairs: array of const): variant; overload;

*Store some object content into a TBSONVariant betDoc type instance*
- object will be initialized with data supplied two by two, as Name,Value pairs, as expected by the corresponding overloaded BSON() function

**function** BSONVariantFieldSelector(const FieldNames: array of RawUTF8): variant; overload;

*Create a fields selector BSON document from a field names list*
- can be used for the projection parameter of a TMongoRequestQuery, e.g.:
  ```rawutf8
  VariantToJSON(BSONVariantFieldSelector(["a","b","c"]))="{"a":1,"b":1,"c":1}"
  ```

**function** BSONVariantFieldSelector(const FieldNamesCSV: RawUTF8): variant; overload;

*Create a fields selector BSON document from a CSV field names list*
- can be used for the projection parameter of a TMongoRequestQuery, e.g.:
  ```rawutf8
  VariantToJSON(BSONVariantFieldSelector('a,b,c'))="{"a":1,"b":1,"c":1}"
  ```
function BSONVariantFromInt64s(const Integers: array of Int64): variant;
    Store an array of 64 bit integer into a TBSONVariant betArray type instance
    - object will be initialized with data supplied e.g. as a TIntegerDynArray

function BSONVariantFromIntegers(const Integers: array of integer): variant;
    Store an array of integer into a TBSONVariant betArray type instance
    - object will be initialized with data supplied e.g. as a TIntegerDynArray

function JavaScript(const JS: RawUTF8; const Scope: TBSONDocument): variant;
    Create a TBSONVariant JavaScript and associated scope custom variant type from a supplied code and document
    - will set a BSON element of betJSScope kind

function JavaScript(const JS: RawUTF8): variant; overload;
    Create a TBSONVariant JavaScript custom variant type from a supplied code
    - will set a BSON element of betJS kind

function JSONBufferToBSONArray(JSON: PUTF8Char; out docs: TBSONDocumentDynArray; DoNotTryExtendedMongoSyntax: boolean=false): boolean;
    Store one JSON array into an array of BSON binary
    - since BSON documents are limited to 16 MB by design, this function will allow to process huge data content, as soon as it is provided as array
    - in addition to the JSON RFC specification strict mode, this method will handle some BSON-like extensions, e.g. unquoted field names
    - if DoNotTryExtendedMongoSyntax is FALSE, then MongoDB Shell syntax will be recognized to create BSON custom values - but it will be slightly slower

function JSONBufferToBSONDocument(JSON: PUTF8Char; var doc: TBSONDocument; DoNotTryExtendedMongoSyntax: boolean=false): TBSONElementType;
    Store some object content, supplied as (extended) JSON into BSON binary
    - warning: the supplied JSON buffer will be modified in-place, if necessary: so you should create a temporary copy before calling this function, or call BSON(const JSON: RawUTF8) function instead
    - in addition to the JSON RFC specification strict mode, this method will handle some BSON-like extensions, e.g. unquoted field names
    - if DoNotTryExtendedMongoSyntax is FALSE, then MongoDB Shell syntax will also be recognized to create BSON custom values, like
      new Date()   ObjectId()   MinKey   MaxKey  /<jRegex>/<jOptions>
    see @http://docs.mongodb.org/manual/reference/mongodb-extended-json
    BSON('{id:new ObjectId(),doc:{name:"John",date:ISODate()}}');
    BSON('{name:"John",field:/acme.*corp/i'});
    - will create the BSON binary without any temporary TDocVariant storage
    - will return the kind of BSON document created, i.e. either betDoc or betArray

function NumberDecimal(const Value: currency): variant; overload;
    Create a TBSONVariant Decimal128 from a currency fixed decimal
    - will store internally a TDecimal128 storage, with explcitly 4 decimals
    - if you want to store some floating-point value, use plain BSON double format

function NumberDecimal(const Value: RawUTF8): variant; overload;
    Create a TBSONVariant Decimal128 from some text corresponding to a floating-point number
    - will store internally a TDecimal128 storage
function ObjectID: variant; overload;

Create a TBSONVariant custom variant type containing a BSON Object ID
- will be filled with some unique values, ready to create a new document key
- will store a BSON element of betObjectID kind

function ObjectID(const Hexa: RawUTF8): variant; overload;

Create a TBSONVariant Object ID custom variant type from a supplied text
- will raise an EBSONException if the supplied text is not valid hexadecimal
- will set a BSON element of betObjectID kind

function ToText(op: TMongoOperation): PShortString; overload;

Ready-to-be displayed text of a TMongoOperation item

function ToText(wc: TMongoClientWriteConcern): PShortString; overload;

Ready-to-be displayed text of a TMongoClientWriteConcern item

function ToText(pref: TMongoClientReplicaSetReadPreference): PShortString; overload;

Ready-to-be displayed text of a TMongoClientReplicaSetReadPreference item

function ToText(kind: TBSONElementType): PShortString; overload;

Ready-to-be displayed text of a TBSONElementType value

function ToText(spec: TDecimal128SpecialValue): PShortString; overload;

Ready-to-be displayed text of a TDecimal128SpecialValue

function VariantSaveMongoJSON(const Value: variant; Mode: TMongoJSONMode): RawUTF8;

Convert any kind of BSON/JSON element, encoded as variant, into JSON
- in addition to default modMongoStrict as rendered by VariantSaveJSON(), this function can render the supplied variant with the Mongo Shell syntax or even raw JSON content

Variables implemented in the SynMongoDB unit

BSONVariantType: TBSONVariant;

Global TCustomVariantType used to register BSON variant types
- if you use this unit, both TDocVariant and TBSONVariant custom types will be registered, since they are needed for any MongoDB / BSON process
27.27. SynMustache.pas unit

Purpose: Logic-less mustache template rendering
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the SynMustache unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
</tbody>
</table>

SynMustache class hierarchy

Objects implemented in the SynMustache unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESynMustache</td>
<td>Exception raised during process of a {{mustache}} template</td>
<td>1450</td>
</tr>
<tr>
<td>TSynMustache</td>
<td>Stores one {{mustache}} pre-rendered template</td>
<td>1453</td>
</tr>
<tr>
<td>TSynMustacheContext</td>
<td>Handle {{mustache}} template rendering context, i.e. all values</td>
<td>1451</td>
</tr>
<tr>
<td>TSynMustacheContextVariant</td>
<td>Handle {{mustache}} template rendering context from a custom variant</td>
<td>1452</td>
</tr>
<tr>
<td>TSynMustacheHelper</td>
<td>Used to store a registered Expression Helper implementation</td>
<td>1451</td>
</tr>
<tr>
<td>TSynMustachePartials</td>
<td>Maintain a list of {{mustache}} partials</td>
<td>1452</td>
</tr>
<tr>
<td>TSynMustacheTag</td>
<td>Store a {{mustache}} tag</td>
<td>1451</td>
</tr>
</tbody>
</table>

ESynMustache = class(ESynException)

Exception raised during process of a {{mustache}} template
TSynMustacheTag = record
    Store a {{mustache}} tag
    Kind: TSynMustacheTagKind;
    The kind of the tag
    SectionOppositeIndex: integer;
    The index in Tags[] of the other end of this section
    - either the index of mtSectionEnd for mtSection/mtInvertedSection
    - or the index of mtSection/mtInvertedSection for mtSectionEnd
    TextLen: integer;
    Stores the mtText buffer length
    TextStart: PUTF8Char;
    Points to the mtText buffer start
    - main template's text is not allocated as a separate string during parsing, but will rather be
    copied directly from the template memory
    Value: RawUTF8;
    The tag content, excluding trailing {{ }} and corresponding symbol
    - is not set for mtText nor mtSetDelimiter
TSynMustacheHelper = record
    Used to store a registered Expression Helper implementation
    Event: TSynMustacheHelperEvent;
    The corresponding callback to process the tag
    Name: RawUTF8;
    The Expression Helper name
TSynMustacheContext = class(TObject)
    Handle {{mustache}} template rendering context, i.e. all values
    - this abstract class should not be used directly, but rather any other overridden class
    constructor Create(Owner: TSynMustache; WR: TTextWriter);
    Initialize the rendering context for the given text writer
    property EscapeInvert: boolean read fEscapeInvert write fEscapeInvert;
    Invert the HTML characters escaping process
    - by default, {{value}} will escape value chars, and {{{value}}} won’t
    - set this property to true to force {{value}} NOT to escape HTML chars and {{{value}}} escaping
    chars (may be useful e.g. for code generation)
    property Helpers: TSynMustacheHelpers read fHelpers write fHelpers;
    The registered Expression Helpers, to handle {{helperName value}} tags
    - use TSynMustache.HelperAdd/HelperDelete class methods to manage the list or retrieve
    standard helpers via TSynMustache.HelpersGetStandardList
property OnStringTranslate: TOnStringTranslate read fOnStringTranslate write fOnStringTranslate;
  Access to the {{"English text"}} translation callback

property Writer: TTextWriter read fWriter;
  Read-only access to the associated text writer instance

TSynMustacheContextVariant = class(TSynMustacheContext)
Handle {{mustache}} template rendering context from a custom variant
- the context is given via a custom variant type implementing TSynInvokeableVariantType.Lookup, e.g. TDocVariant or TSMVariant

constructor Create(Owner: TSynMustache; WR: TTextWriter; SectionMaxCount: integer; const aDocument: variant);
  Initialize the context from a custom variant document
  - note that the aDocument instance shall be available during all lifetime of this TSynMustacheContextVariant instance
  - you should not use this constructor directly, but the corresponding TSynMustache.Render*() methods

TSynMustachePartials = class(TObject)
Maintain a list of {{mustache}} partials
- this list of partials template could be supplied to TSynMustache.Render() method, to render {{>partials}} as expected
- using a dedicated class allows to share the partials between execution context, without recurring to non SOLID global variables
- you may also define "internal" partials, e.g. {{<foo}}This is foo{{/foo}}

constructor Create; overload;
  Initialize the template partials storage
  - after creation, the partials should be registered via the Add() method
  - you shall manage this instance life time with a try..finally Free block

constructor CreateOwned(const NameTemplatePairs: array of RawUTF8); overload;
  Initialize a template partials storage with the supplied templates
  - partials list is expected to be supplied in Name / Template pairs
  - this instance can be supplied as parameter to the TSynMustache.Render() method, which will free the instances as soon as it finishes

destructor Destroy; override;
  Delete the partials

function Add(const aName: RawUTF8; aTemplateStart,aTemplateEnd: PUTF8Char): TSynMustache; overload;
  Register a {{>partialName}} template
  - returns the parsed template

function Add(const aName,aTemplate: RawUTF8): TSynMustache; overload;
  Register a {{>partialName}} template
  - returns the parsed template
**class function** CreateOwned(const Partials: variant): TSynMustachePartials; overload;

*Initialize a template partials storage with the supplied templates*
- partials list is expected to be supplied as a dvObject TDocVariant, each member being the name/template string pairs
- if the supplied variant is not a matching TDocVariant, will return nil
- this instance can be supplied as parameter to the TSynMustache.Render() method, which will free the instances as soon as it finishes

**function** FoundInTemplate(const text: RawUTF8):_PTRInt;

*Search some text withing the {{mustache}} partial*

**property** List: TRawUTF8List read fList;

*Low-level access to the internal partials list*

---

**TSynMustache = class(TObject)**

*Stores one {{mustache}} pre-rendered template*
- once parsed, a template will be stored in this class instance, to be rendered lated via the Render() method
- you can use the Parse() class function to maintain a shared cache of parsed templates
- implements all official mustache specifications, and some extensions
- handles {{}.}} pseudo-variable for the current context object (very handy when looping through a simple list, for instance)
- handles {{-index}} pseudo-variable for the current context array index (1-based value) so that e.g. "My favorite things:

> {{#things}}

> 1. Peanut butter

> 2. Pen spinning

> 3. Handstands"

renders as "My favorite things:

> 1. Peanut butter

> 2. Pen spinning

> 3. Handstands"

- you could use {{-index0}} for 0-based index value
- handles -first -last and -odd pseudo-section keys, e.g. "{{#things}}{{^-first}}, {{/-first}}{{.}}{{/things}}" over {things: ["Peanut butter", "Pen spinning", "Handstands"]} renders as "My favorite things:

> 1. Peanut butter

> 2. Pen spinning

> 3. Handstands"

- allows inlined partial templates, to be defined e.g. as {{<foo}}This is the foo partial {{myValue}}template{{/foo}}
- features {{"English text"}} translation, via a custom callback
- this implementation is thread-safe and re-entrant (i.e. the same TSynMustache instance can be used by several threads at once)

**constructor** Create(const aTemplate: RawUTF8); overload;

*Initialize and parse a pre-rendered {{mustache}} template*
- you should better use the Parse() class function instead, which features an internal thread-safe cache

**constructor** Create(aTemplate: PUTF8Char; aTemplateLen: integer); overload; virtual;

*Initialize and parse a pre-rendered {{mustache}} template*
- you should better use the Parse() class function instead, which features an internal thread-safe cache

**destructor** Destroy; override;

*Finalize internal memory*

**function** FoundInTemplate(const text: RawUTF8): boolean;

*Search some text within the {{mustache}} template text*
```pascal
class function HelperFind(const Helpers: TSynMustacheHelpers; aName: PUTF8Char; aNameLen: integer): integer;
  
  Search for one Expression Helper event by name

class function HelpersGetStandardList: TSynMustacheHelpers; overload;
  
  Returns a list of most used static Expression Helpers
  - registered helpers are DateTimeToText, DateToText, DateFmt, TimeLogToText, BlobToBase64, JSONQuote, JSONQuoteURI, ToJSON, EnumTrim, EnumTrimRight, Lower, Upper, PowerOfTwo, Equals (expecting two parameters), MarkdownToHtml, SimpleToHtml (Markdown with no HTML pass-through) and WikiToHtml (following TTextWriter.AddHtmlEscapeWiki syntax)
  - an additional #if helper is also registered, which would allow runtime view logic, via = <> <= >= operators over two values:
    {{#if .,"=",123}}  {{#if Total,">",1000}}  {{#if info,"<>",""}}
    which may be shortened as such:
    {{#if .=123}}  {{#if Total>1000}}  {{#if info<>""}}

class function HelpersGetStandardList(const aNames: array of RawUTF8; const aEvents: array of TSynMustacheHelperEvent): TSynMustacheHelpers; overload;
  
  Returns a list of most used static Expression Helpers, adding some custom callbacks
  - is just a wrapper around HelpersGetStandardList and HelperAdd()

class function Parse(const aTemplate: RawUTF8): TSynMustache;
  
  Parse a {{mustache}} template, and returns the corresponding TSynMustache instance
  - an internal cache is maintained by this class function
  - this implementation is thread-safe and re-entrant: i.e. the same TSynMustache returned instance can be used by several threads at once
  - will raise an ESynMustache exception on error
```
function Render(const Context: variant; Partialials: TSynMustachePartials=nil; Helpers: TSynMustacheHelpers=nil; OnTranslate: TOnStringTranslate=nil; EscapeInvert: boolean=false): RawUTF8;

Renders the {{mustache}} template from a variant defined context
- the context is given via a custom variant type implementing
TSynInvokeableVariantType.Lookup, e.g. TDocVariant or TSMVariant
- you can specify a list of partials via TSynMustachePartials.CreateOwned, a list of Expression
Helpers, or a custom {{"English text"}} callback
- can be used e.g. via a TDocVariant:
var mustache := TSynMustache;
doc: variant;
html: RawUTF8;
begin
mustache := TSynMustache.Parse(
'Hello {{name}}'#{13}#{10}'You have just won {{value}} dollars!');
TDocVariant.New(doc);
doc.name := 'Chris';
doc.value := 10000;
html := mustache.Render(doc);
// here html='Hello Chris'#{13}#{10}'You have just won 10000 dollars!'
- you can also retrieve the context from an ORM query of mORMot.pas:
dummy := TSynMustache.Parse(
'{{#items}}'#{13}#{10}'{{Int}}={{Test}}'#{13}#{10}'{{/items}}').Render(
aClient.RetrieveDocVariantArray(TSQLRecordTest,'items', 'Int,Test');?></aClient.RetrieveDocVariantArray(TSQLRecordTest,'items', 'Int,Test'));
- set EscapeInvert = true to force {{value}} NOT to escape HTML chars and {{{value}} escaping
chars (may be useful e.g. for code generation)

function RenderJSON(const JSON: RawUTF8; const Args,Params: array of const; Partialials: TSynMustachePartials=nil; Helpers: TSynMustacheHelpers=nil; OnTranslate: TOnStringTranslate=nil; EscapeInvert: boolean=false): RawUTF8; overload;

Renders the {{mustache}} template from JSON defined context
- the context is given via a JSON object, defined with parameters
- you can specify a list of partials via TSynMustachePartials.CreateOwned, a list of Expression
Helpers, or a custom {{"English text"}} callback
- is just a wrapper around Render(_JsonFastFmt())
- you can write e.g. with the extended JSON syntax:
  html := mustache.RenderJSON(
'{{name:?,value:?}}',[],['Chris',10000]);
- set EscapeInvert = true to force {{value}} NOT to escape HTML chars and {{{value}} escaping
chars (may be useful e.g. for code generation)

function RenderJSON(const JSON: RawUTF8; Partialials: TSynMustachePartials=nil; Helpers: TSynMustacheHelpers=nil; OnTranslate: TOnStringTranslate=nil; EscapeInvert: boolean=false): RawUTF8; overload;

Renders the {{mustache}} template from JSON defined context
- the context is given via a JSON object, defined from UTF-8 buffer
- you can specify a list of partials via TSynMustachePartials.CreateOwned, a list of Expression
Helpers, or a custom {{"English text"}} callback
- is just a wrapper around Render(_JsonFast())
- you can write e.g. with the extended JSON syntax:
  html := mustache.RenderJSON(
'{things: ["one", "two", "three"]}');
- set EscapeInvert = true to force {{value}} NOT to escape HTML chars and {{{value}} escaping
chars (may be useful e.g. for code generation)
class function TryRenderJson(const aTemplate, aJSON: RawUTF8; out aContent: RawUTF8): boolean;

Parse and render a {{mustache}} template over the supplied JSON
- an internal templates cache is maintained by this class function
- returns TRUE and set aContent the rendered content on success
- returns FALSE if the template is not correct

class function UnParse(const aTemplate: RawUTF8): boolean;

Remove the specified {{mustache}} template from the internal cache
- returns TRUE on success, or FALSE if the template was not cached by a previous call to Parse()
class function

class procedure HelperAdd(var Helpers: TSynMustacheHelpers; const aName: RawUTF8; aEvent: TSynMustacheHelperEvent); overload;

Register one Expression Helper callback for a given list of helpers
- i.e. to let aEvent process {{aName value}} tags
- the supplied name will be checked against the current list, and replace any existing entry

class procedure HelperAdd(var Helpers: TSynMustacheHelpers; const aNames: array of RawUTF8; const aEvents: array of TSynMustacheHelperEvent); overload;

Register several Expression Helper callbacks for a given list of helpers
- the supplied names will be checked against the current list, and replace any existing entry

class procedure HelperDelete(var Helpers: TSynMustacheHelpers; const aName: RawUTF8);

Unregister one Expression Helper callback for a given list of helpers

procedure RenderContext(Context: TSynMustacheContext; TagStart, TagEnd: integer; Partials: TSynMustachePartials; NeverFreePartials: boolean);

Renders the {{mustache}} template into a destination text buffer
- the context is given via our abstract TSynMustacheContext wrapper
- the rendering extended in fTags[] is supplied as parameters
- you can specify a list of partials via TSynMustachePartials.CreateOwned

property SectionMaxCount: Integer read fSectionMaxCount;

The maximum possible number of nested contexts

property Template: RawUTF8 read fTemplate;

Read-only access to the raw {{mustache}} template content

Types implemented in the SynMustache unit

TSynMustacheHelperEvent = procedure(const Value: variant; out result: variant) of object;

Callback signature used to process an Expression Helper variable
- i.e. {{helperName value}} tags
- returned value will be used to process as replacement of a single {{tag}}

TSynMustacheHelpers = array of TSynMustacheHelper;

Used to store all registered Expression Helpers
- i.e. {{helperName value}} tags
- use TSynMustache.HelperAdd/HelperDelete class methods to manage the list or retrieve standard helpers via TSynMustache.HelpersGetStandardList
```
TSynMustacheSectionType = ( msNothing, msSingle, msSinglePseudo, msList );

States the section content according to a given value
- msNothing for false values or empty lists
- msSingle for non-false values but not a list
- msSinglePseudo is for *-first *-last *-odd and helper values
- msList for non-empty lists

TSynMustacheTagDynArray = array of TSynMustacheTag;
Save all {{mustache}} tags of a given template

TSynMustacheTagKind = ( mtVariable, mtVariableUnescape, mtVariableUnescapeAmp,
mtSection, mtInvertedSection, mtSectionEnd, mtComment, mtPartial, mtSetPartial,
mtSetDelimiter, mtTranslate, mtText );
Identify the {{mustache}} tag kind
- mtVariable if the tag is a variable - e.g. {{myValue}} - or an Expression Helper - e.g. {{helperName valueName}}
- mtVariableUnescape, mtVariableUnescapeAmp to unescape the variable HTML - e.g. {{myRawValue}} or {{name}}
- mtSection and mtInvertedSection for sections beginning - e.g. {{#person}} or {{^person}}
- mtSectionEnd for sections ending - e.g. {{/person}}
- mtComment for comments - e.g. {{! ignore me}}
- mtPartial for partials - e.g. {{> next_more}}
- mtSetPartial for setting an internal partial - e.g. {{<foo}}This is the foo partial {{myValue}} template{{/foo}}
- mtSetDelimiter for setting custom delimeter symbols - e.g. {{=<% %>=}} - Warning: current implementation only supports two character delimiters
- mtTranslate for content i18n via a callback - e.g. {{"English text"}}
- mtText for all text that appears outside a symbol
```

**Constants implemented in the SynMustache unit**

```
NULL_OR_COMMA: array[boolean] of RawUTF8 = ('null', '', '');
This constant can be used to define as JSON a tag value as separator

NULL_OR_TRUE: array[boolean] of RawUTF8 = ('null', 'true');
This constant can be used to define as JSON a tag value
```
27.28. SynOleDB.pas unit

_Purpose:_ Fast OleDB direct access classes
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the _SynOleDB_ unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>SynCommons</em></td>
<td>Common functions used by most Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
<tr>
<td><em>SynDB</em></td>
<td>Abstract database direct access classes - this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; 1.18</td>
<td>1204</td>
</tr>
<tr>
<td><em>SynLog</em></td>
<td>Logging functions used by Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1363</td>
</tr>
<tr>
<td><em>SynTable</em></td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features - as a complement to SynCommons, which tended to increase too much - licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1721</td>
</tr>
</tbody>
</table>
**Objects implemented in the SynOleDb unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOleDbException</td>
<td>Generic Exception type, generated for OleDB connection</td>
<td>1462</td>
</tr>
<tr>
<td>IAccessor</td>
<td>Provides methods for accessor management, to access OleDB data</td>
<td>1462</td>
</tr>
<tr>
<td>IColumnsInfo</td>
<td>Expose information about columns of an OleDB rowset or prepared command</td>
<td>1462</td>
</tr>
<tr>
<td>ICommand</td>
<td>Provide methods to execute commands</td>
<td>1461</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>ICommandText</td>
<td>Methods to access the ICommand text to be executed</td>
<td>1461</td>
</tr>
<tr>
<td>IDataInitialize</td>
<td>Create an OleDB data source object using a connection string</td>
<td>1461</td>
</tr>
<tr>
<td>IDbCreateCommand</td>
<td>Used on an OleDB session to obtain a new command</td>
<td>1462</td>
</tr>
<tr>
<td>IDbCreateSession</td>
<td>Obtain a new session to a given OleDB data source</td>
<td>1461</td>
</tr>
<tr>
<td>IDbInitialize</td>
<td>Packed records initialize and uninitialize OleDB data source objects and enumerators</td>
<td>1461</td>
</tr>
<tr>
<td>IDbPromptInitialize</td>
<td>Allows the display of the data link dialog boxes programmatically</td>
<td>1462</td>
</tr>
<tr>
<td>IDbSchemaRowset</td>
<td>Used to retrieve the database metadata (e.g. tables and fields layout)</td>
<td>1462</td>
</tr>
<tr>
<td>IErrorRecords</td>
<td>Interface used to retrieve enhanced custom error information</td>
<td>1462</td>
</tr>
<tr>
<td>IRowset</td>
<td>Provides methods for fetching rows sequentially, getting the data from those rows, and managing rows</td>
<td>1461</td>
</tr>
<tr>
<td>ITransaction</td>
<td>Commit, abort, and obtain status information about OleDB transactions</td>
<td>1461</td>
</tr>
<tr>
<td>ITransactionLocal</td>
<td>Optional interface on OleDB sessions, used to start, commit, and abort transactions on the session</td>
<td>1461</td>
</tr>
<tr>
<td>ITransactionOptions</td>
<td>Gets and sets a suite of options associated with an OleDB transaction</td>
<td>1461</td>
</tr>
<tr>
<td>TBoId</td>
<td>Packed records</td>
<td>1461</td>
</tr>
<tr>
<td>TOleDBACEConnectionProperties</td>
<td>OleDB connection properties to Microsoft Access Database</td>
<td>1464</td>
</tr>
<tr>
<td>TOleDBAS400ConnectionProperties</td>
<td>OleDB connection properties to IBM AS/400</td>
<td>1464</td>
</tr>
<tr>
<td>TOleDBConnection</td>
<td>Implements an OleDB connection</td>
<td>1465</td>
</tr>
<tr>
<td>TOleDBConnectionPropert ies</td>
<td>Will implement properties shared by OleDB connections</td>
<td>1462</td>
</tr>
<tr>
<td>TOleDBInformixConnectionProperties</td>
<td>OleDB connection properties to Informix Server</td>
<td>1465</td>
</tr>
<tr>
<td>TOleDBJetConnectionProperties</td>
<td>Jet is not available on Win64 OleDB connection properties to Jet/MSAccess .mdb files</td>
<td>1464</td>
</tr>
<tr>
<td>TOleDBMSOracleConnectionProperties</td>
<td>OleDB connection properties to an Oracle database using Microsoft's Provider</td>
<td>1463</td>
</tr>
<tr>
<td>TOleDBMSQL2005ConnectionProperties</td>
<td>OleDB connection properties to Microsoft SQL Server 2005, via SQL Server Native Client (SQL Server 2005)</td>
<td>1464</td>
</tr>
<tr>
<td>TOleDBMSQL2012ConnectionProperties</td>
<td>OleDB connection properties to Microsoft SQL Server 2008/2012, via SQL Server Native Client 11.0 (Microsoft SQL Server 2012 Native Client)</td>
<td>1464</td>
</tr>
<tr>
<td>TOleDBMSSQLConnectionProperties</td>
<td>OleDB connection properties to Microsoft SQL Server 2008-2012, via SQL Server Native Client 10.0 (SQL Server 2008)</td>
<td>1464</td>
</tr>
</tbody>
</table>
Objects | Description | Page
---|---|---
T OleDBMySQLConnectionProperties | OleDB connection properties to MySQL Server | 1464
T OleDBODBCSQLConnectionProperties | OleDB connection properties via Microsoft Provider for ODBC | 1465
T OleDBOracleConnectionProperties | OleDB connection properties to an Oracle database using Oracle's Provider | 1463
T OleDBStatement | Implements an OleDB SQL query statement | 1467
T OleDBStatementParam | Used to store properties and value about one T OleDBStatement Param | 1466

TBoid = record

Packed records

IDBInitialize = interface(IUnknown)

Packed records initialize and uninitialize OleDB data source objects and enumerators

IDataInitialize = interface(IUnknown)

Create an OleDB data source object using a connection string

IDBCreateSession = interface(IUnknown)

Obtain a new session to a given OleDB data source

ITransaction = interface(IUnknown)

Commit, abort, and obtain status information about OleDB transactions

ITransactionOptions = interface(IUnknown)

Gets and sets a suite of options associated with an OleDB transaction

ITransactionLocal = interface(ITransaction)

Optional interface on OleDB sessions, used to start, commit, and abort transactions on the session

ICommand = interface(IUnknown)

Provide methods to execute commands

ICommandText = interface(ICommand)

Methods to access the ICommand text to be executed

IRowset = interface(IUnknown)

Provides methods for fetching rows sequentially, getting the data from those rows, and managing rows

function AddRefRows(cRows: PPtrUInt; rghRows: PPtrUIntArray; rgRefCounts, rgRowStatus: PCardinalArray): HRESULT; stdcall;

Adds a reference count to an existing row handle

function GetData(HROW: HROW; HACCESSOR: HACCESSOR; pData: Pointer): HRESULT; stdcall;

Retrieves data from the rowset's copy of the row
function GetNextRows(hReserved: HCHAPTER; lRowsOffset: PTrInt; cRows: PTrInt; out pRowsObtained: PTrUint; var prghRows: pointer): HResult; stdcall;

  Fetches rows sequentially, remembering the previous position
  - this method has been modified from original OleDB.pas to allow direct typecast of prghRows parameter to pointer(fRowStepHandles)

function ReleaseRows(cRows: UINT; rghRows: PPtrUIntArray; rgRowOptions, rgRefCount, rgRowStatus: PCardinalArray): HResult; stdcall;

  Releases rows

function RestartPosition(hReserved: HCHAPTER): HResult; stdcall;

  Repositions the next fetch position to its initial position
  - that is, its position when the rowset was first created

IErrorRecords = interface(IUnknown)
  Interface used to retrieve enhanced custom error information

IDBCreateCommand = interface(IUnknown)
  Used on an OleDB session to obtain a new command

IAccessor = interface(IUnknown)
  Provides methods for accessor management, to access OleDB data
  - An accessor is a data structure created by the consumer that describes how row or parameter data from the data store is to be laid out in the consumer's data buffer.
  - For each column in a row (or parameter in a set of parameters), the accessor contains a binding. A binding is a DBBinding data structure that holds information about a column or parameter value, such as its ordinal value, data type, and destination in the consumer's buffer.

IColumnsInfo = interface(IUnknown)
  Expose information about columns of an OleDB rowset or prepared command

IDBPromptInitialize = interface(IUnknown)
  Allows the display of the data link dialog boxes programmatically

IDBSchemaRowset = interface(IUnknown)
  Used to retrieve the database metadata (e.g. tables and fields layout)

EOleDBException = class(ESQLDBException)
  Generic Exception type, generated for OleDB connection

TOLedBCOnnectionProperties = class(TSQLDBConnectionPropertiesThreadSafe)
  Will implement properties shared by OleDB connections

function ColumnTypeNativeToDB(const aNativeType: RawUTF8; aScale: integer): TSQLDBFieldType; override;
  Convert a textual column data type, as retrieved e.g. from SQLGetField, into our internal primitive types

function ConnectionStringDialogExecute(Parent: HWND=0): boolean;
  Display the OleDB/ADO Connection Settings dialog to customize the OleDB connection string
  - returns TRUE if the connection string has been modified
  - Parent is an optional GDI Window Handle for modal display
function NewConnection: TSQLDBConnection; override;

Create a new connection
- call this method if the shared MainConnection is not enough (e.g. for multi-thread access)
- the caller is responsible of freeing this instance
- this overridden method will create an TOleDBConnection instance

procedure GetFields(const aTableName: RawUTF8; out Fields: TSQLDBColumnDefineDynArray); override;

Retrieve the column/field layout of a specified table
- will retrieve the corresponding metadata from OleDB interfaces if SQL direct access was not defined

procedure GetTableNames(out Tables: TRawUTF8DynArray); override;

Get all table names
- will retrieve the corresponding metadata from OleDB interfaces if SQL direct access was not defined

property ConnectionString: SynUnicode read fConnectionString write fConnectionString;

The associated OleDB connection string
- is set by the Create() constructor most of the time from the supplied server name, user id and password, according to the database provider corresponding to the class
- you may want to customize it via the ConnectionStringDialogExecute method, or to provide some additional parameters

property OnCustomError: TOleDBOnCustomError read fOnCustomError write fOnCustomError;

Custom Error handler for OleDB COM objects
- returns TRUE if specific error was retrieved and has updated ErrorMessage and InfoMessage
- default implementation just returns false

property ProviderName: RawUTF8 read fProviderName;

The associated OleDB provider name, as set for each class

TOleDBOracleConnectionProperties = class(TOleDBConnectionProperties)

OleDb connection properties to an Oracle database using Oracle's Provider
- this will use the native OleDB provider supplied by Oracle see @http://download.oracle.com/docs/cd/E11882_01/win.112/e17726/toc.htm

TOleDBMSOracleConnectionProperties = class(TOleDBOracleConnectionProperties)

OleDb connection properties to an Oracle database using Microsoft's Provider
- this will use the generic (older) OleDB provider supplied by Microsoft which would not be used any more: "This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use Oracle's OLE DB provider." see http://msdn.microsoft.com/en-us/library/ms675851
T OleDBMSSQLConnectionProperties = class(TOleDbConnectionProperties)
OleDb connection properties to Microsoft SQL Server 2008-2012, via SQL Server Native Client 10.0 (SQL Server 2008)
- this will use the native OleDB provider supplied by Microsoft see
- is aUserID="" at Create, it will use Windows Integrated Security for the connection
- will use the SQLNCLI10 provider, which will work on Windows XP; if you want all features,
  especially under MS SQL 2012, use the inherited class TOleDbMSSQL2012ConnectionProperties; if,
  on the other hand, you need to connect to a old MS SQL Server 2005, use
  TOleDbMSSQL2005ConnectionProperties, or set your own provider string

T OleDBMSSQL2005ConnectionProperties = class(TOleDbConnectionProperties)
OleDb connection properties to Microsoft SQL Server 2005, via SQL Server Native Client (SQL Server 2005)
- this overridden version will use the SQLNCLI provider, which is deprecated but may be an
  alternative with MS SQL Server 2005
- is aUserID="" at Create, it will use Windows Integrated Security for the connection

   constructor Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8); override;
   Initialize the connection properties
   - this overridden version will disable the MultipleValuesInsert() optimization as defined
     in TSQLDBConnectionProperties.Create(), since INSERT with multiple VALUES (..),(..),(..) is available
     only since SQL Server 2008

T OleDBMSSQL2012ConnectionProperties = class(TOleDbMSSQLConnectionProperties)
OleDb connection properties to Microsoft SQL Server 2008/2012, via SQL Server Native Client 11.0
(Microsoft SQL Server 2012 Native Client)
  package corresponding to your Operating System: note that the "X64 Package" will also install the
  32-bit version of the client
- this overridden version will use newer SQLNCLI11 provider, but won't work under Windows XP -
  in this case, it will fall back to SQLNCLI10 - see
- if aUserID="" at Create, it will use Windows Integrated Security for the connection
- for SQL Express LocalDB edition, just use aServerName='(localdb)\v11.0'

T OleDBMySQLConnectionProperties = class(TOleDbConnectionProperties)
OleDb connection properties to MySQL Server

T OleDBJetConnectionProperties = class(TOleDbConnectionProperties)
Jet is not available on Win64 OleDb connection properties to Jet/MSAccess .mdb files
- the server name should be the .mdb file name
- note that the Jet OleDB driver is not available under Win64 platform

T OleDBACEConnectionProperties = class(TOleDbConnectionProperties)
OleDb connection properties to Microsoft Access Database

T OleDBAS400ConnectionProperties = class(TOleDbConnectionProperties)
OleDb connection properties to IBM AS/400
T OleDBInformixConnectionProperties = class(TOleDbConnectionProperties)

OleDb connection properties to Informix Server

T OleDBODBCSQLConnectionProperties = class(TOleDbConnectionProperties)

OleDb connection properties via Microsoft Provider for ODBC
- this will use the ODBC provider supplied by Microsoft see
- an ODBC Driver should be specified at creation
- you should better use direct connection classes, like TOleDbMSSQLConnectionProperties or
TOleDbOracleConnectionProperties as defined in SynDBODBC.pas

constructor Create(const aDriver, aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8); reintroduce;

Initialize the properties
- an additional parameter is available to set the ODBC driver to use
- you may also set aDriver="" and modify the connection string directly, e.g. adding '{ DSN=name
| FileDSN=filename }';

property Driver: RawUTF8 read fDriver;

The associated ODBC Driver name, as specified at creation

T OleDBConnection = class(TSQLDBConnectionThreadSafe)

Implements an OleDb connection
- will retrieve the remoteDataBase behavior from a supplied TSQLDBConnectionProperties class,
shared among connections

constructor Create(aProperties: TSQLDBConnectionProperties); override;

Connect to a specified OleDb database

destructor Destroy; override;

Release all associated memory and OleDb COM objects

function IsConnected: boolean; override;

Return TRUE if Connect has been already successfully called

function NewStatement: TSQLDBStatement; override;

Initialize a new SQL query statement for the given connection
- the caller should free the instance after use

procedure Commit; override;

Commit changes of a Transaction for this connection
- StartTransaction method must have been called before

procedure Connect; override;

Connect to the specified database
- should raise an EOleDbException on error

procedure Disconnect; override;

Stop connection to the specified database
- should raise an EOleDbException on error
procedure Rollback; override;

Discard changes of a Transaction for this connection
- StartTransaction method must have been called before

procedure StartTransaction; override;

Begin a Transaction for this connection
- be aware that not all OleDB provider support nested transactions see

property OleDBErrorMessage: string read fOleDBErrorMessage;

Internal error message, as retrieved from the OleDB provider

property OleDBInfoMessage: string read fOleDBInfoMessage;

Internal information message, as retrieved from the OleDB provider

property OleDBProperties: TOleDBConnectionProperties read fOleDBProperties;

The associated OleDB database properties

TOleDBStatementParam = record

Used to store properties and value about one TOleDBStatement Param
- we don't use a Variant, not the standard TSQLDBParam record type, but manual storage for
better performance
- whole memory block of a TOleDBStatementParamDynArray will be used as the source Data for
the OleDB parameters - so we should align data carefully packed records

VArray: TRawUTF8DynArray;

Storage used for table variables

VBlob: RawByteString;

Storage used for BLOB (ftBlob) values
- will be refered as DBTYPE_BYREF when sent as OleDB parameters, to avoid unnecessary
memory copy

VFill: array[sizeof(TSQLDBFieldType)+sizeof(TSQLDBParamInOutType)+sizeof(integer)..<SizeOf(Int64)-1] of byte;

So that VInt64 will be 8 bytes aligned

VInOut: TSQLDBParamInOutType;

Define if parameter can be retrieved after a stored procedure execution

VInt64: Int64;

Storage used for ftInt64, ftDouble, ftDate and ftCurrency value

VIUnknown: IUnknown;

Storage used for table variables

VStatus: integer;

Storage used for the OleDB status field
- if VStatus=ord(stsIsNull), then it will bind a NULL with the type as set by VType (to avoid
conversion error like in [e8c211062e])
VText: WideString;

*Storage used for TEXT (ftUTF8) values*
- we store TEXT here as WideString, and not RawUTF8, since OleDB expects the text to be provided with Unicode encoding
- for some providers (like Microsoft SQL Server 2008 R2, AFAIK), using DBTYPE_WSTR value (i.e. what the doc. says) will raise an OLEDB Error 80040E1D (DB_E_UNSUPPORTEDCONVERSION, i.e. 'Requested conversion is not supported'): we found out that only DBTYPE_BSTR type (i.e. OLE WideString) does work... so we'll use it here! Shame on Microsoft!
- what's fine with DBTYPE_BSTR is that it can be resized by the provider in case of VInOut in [paramOut, paramInOut] - so let it be

VType: TSQLDBFFieldType;

*The column/parameter Value type*

T OleDBStatement = class(TSQLDBStatement)

*Implements an OleDB SQL query statement*
- this statement won't retrieve all rows of data, but will allow direct per-row access using the Step() and Column*() methods

**constructor** Create(aConnection: TSQLDBConnection); **override**;

*Create an OleDb statement instance, from an OleDb connection*
- the Execute method can be called only once per TOleDBStatement instance
- if the supplied connection is not of TOleDBConnection type, will raise an exception

**destructor** Destroy; **override**;

*Release all associated memory and COM objects*

**function** ColumnBlob(Col: integer): RawByteString; **override**;

*Return a Column as a blob value of the current Row, first Col is 0*
- ColumnBlob() will return the binary content of the field is was not ftBlob, e.g. a 8 bytes RawByteString for a vtInt64/vtDouble/vtDate/vtCurrency, or a direct mapping of the RawUnicode

**function** ColumnCurrency(Col: integer): currency; **override**;

*Return a Column currency value of the current Row, first Col is 0*
- should retrieve directly the 64 bit Currency content, to avoid any rounding/conversion error from floating-point types

**function** ColumnDateTime(Col: integer): TDateTime; **override**;

*Return a Column date and time value of the current Row, first Col is 0*

**function** ColumnDouble(Col: integer): double; **override**;

*Return a Column floating point value of the current Row, first Col is 0*

**function** ColumnIndex(const aColumnName: RawUTF8): integer; **override**;

*Returns the Column index of a given Column name*
- Columns numeration (i.e. Col value) starts with 0
- returns -1 if the Column name is not found (via case insensitive search)

**function** ColumnInt(Col: integer): Int64; **override**;

*Return a Column integer value of the current Row, first Col is 0*
function ColumnName(Col: integer): RawUTF8; override;

Retrieve a column name of the current Row
- Columns numeration (i.e. Col value) starts with 0
- it's up to the implementation to ensure than all column names are unique

function ColumnNull(Col: integer): boolean; override;

Returns TRUE if the column contains NULL

function ColumnString(Col: integer): string; override;

Return a Column text generic VCL string value of the current Row, first Col is 0

function ColumnToVariant(Col: integer; var Value: Variant): TSQLDBFieldType; override;

Return a Column as a variant
- this implementation will retrieve the data with no temporary variable (since TQuery calls this method a lot, we tried to optimize it)
- a ftUTF8 content will be mapped into a generic WideString variant for pre-Unicode version of Delphi, and a generic UnicodeString (=string) since Delphi 2009: you may not loose any data during charset conversion
- a ftBlob content will be mapped into a T BlobData AnsiString variant

function ColumnType(Col: integer; FieldSize: PInteger=nil): TSQLDBFieldType; override;

The Column type of the current Row
- ftCurrency type should be handled specifically, for faster process and avoid any rounding issue, since currency is a standard OleDB type
- FieldSize can be set to store the size in chars of a ftUTF8 column (0 means BLOB kind of TEXT column)

function ColumnUTF8(Col: integer): RawUTF8; override;

Return a Column UTF-8 encoded text value of the current Row, first Col is 0

function ParamToVariant(Param: Integer; var Value: Variant; CheckIsOutParameter: boolean=true): TSQLDBFieldType; override;

Retrieve the parameter content, after SQL execution
- the leftmost SQL parameter has an index of 1
- to be used e.g. with stored procedures
- any TEXT parameter will be retrieved as WideString Variant (i.e. as stored in TOleDBStatementParam)

function Step(SeekFirst: boolean=false): boolean; override;

After a statement has been prepared via Prepare() + ExecutePrepared() or Execute(), this method must be called one or more times to evaluate it
- you shall call this method before calling any Column*() methods
- return TRUE on success, with data ready to be retrieved by Column*()
- return FALSE if no more row is available (e.g. if the SQL statement is not a SELECT but an UPDATE or INSERT command)
- access the first or next row of data from the SQL Statement result: if SeekFirst is TRUE, will put the cursor on the first row of results, otherwise, it will fetch one row of data, to be called within a loop
- raise an ESQLEOleDBException on any error
function UpdateCount: integer; override;

Gets a number of updates made by latest executed statement

procedure Bind(Param: Integer; Value: Int64; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind an integer value to a parameter
- the leftmost SQL parameter has an index of 1
- raise an EOleDBException on any error

procedure Bind(Param: Integer; Value: double; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a double value to a parameter
- the leftmost SQL parameter has an index of 1
- raise an EOleDBException on any error

procedure BindArray(Param: Integer; const Values: array of RawUTF8); overload; override;

Bind a array of RawUTF8 (255 length max) values to a parameter
- using TABLE variable (MSSQL 2008 & UP). Must be created in the database as:
  CREATE TYPE dbo.StrList AS TABLE(id nvarchar(255) NULL)
- must be declared in the database

procedure BindArray(Param: Integer; const Values: array of Int64); overload; override;

Bind an array of Int64 values to a parameter
- using TABLE variable (MSSQL 2008 & UP). Must be created in the database as:
  CREATE TYPE dbo.IDList AS TABLE(id bigint NULL)
- Internally BindArray(0, [1, 2,3]) is the same as:
  declare @a dbo.IDList;
  insert into @a (id) values (1), (2), (3);
  SELECT usr.ID   FROM user usr WHERE usr.ID IN (select id from @a)

procedure BindBlob(Param: Integer; Data: pointer; Size: integer; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a Blob buffer to a parameter
- the leftmost SQL parameter has an index of 1
- raise an EOleDBException on any error

procedure BindBlob(Param: Integer; const Data: RawByteString; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a Blob buffer to a parameter
- the leftmost SQL parameter has an index of 1
- raise an EOleDBException on any error

procedure BindCurrency(Param: Integer; Value: currency; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a currency value to a parameter
- the leftmost SQL parameter has an index of 1
- raise an EOleDBException on any error
procedure BindDateTime(Param: Integer; Value: TDateTime; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a TDateTime value to a parameter
- the leftmost SQL parameter has an index of 1
- raise an EOleDBException on any error

procedure BindNull(Param: Integer; IO: TSQLDBParamInOutType=paramIn; BoundType: TSQLDBFieldType=ftNull); override;

Bind a NULL value to a parameter
- the leftmost SQL parameter has an index of 1
- OleDB during MULTI INSERT statements expect BoundType to be set in TOleDbStatementParam, and its VStatus set to ord(stIsNull)
- raise an EOleDBException on any error

procedure BindTextP(Param: Integer; Value: PUTF8Char; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a UTF-8 encoded buffer text (#0 ended) to a parameter
- the leftmost SQL parameter has an index of 1
- raise an EOleDBException on any error

procedure BindTextS(Param: Integer; const Value: string; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a VCL string to a parameter
- the leftmost SQL parameter has an index of 1
- raise an EOleDBException on any error

procedure BindTextU(Param: Integer; const Value: RawUTF8; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind a UTF-8 encoded string to a parameter
- the leftmost SQL parameter has an index of 1
- raise an EOleDBException on any error

procedure BindTextW(Param: Integer; const Value: WideString; IO: TSQLDBParamInOutType=paramIn); overload; override;

Bind an OLE WideString to a parameter
- the leftmost SQL parameter has an index of 1
- raise an EOleDBException on any error

procedure ColumnsToJSON(WR: TJSONWriter); override;

Append all columns values of the current Row to a JSON stream
- will use WR.Expand to guess the expected output format
- fast overridden implementation with no temporary variable
- BLOB field value is saved as Base64, in the "\uFFF0base64encodedbinary" format and contains true BLOB data

procedure ExecutePrepared; override;

Execute an UTF-8 encoded SQL statement
- parameters marked as ? should have been already bound with Bind*() functions above
- raise an EOleDBException on any error
procedure FromRowSet(RowSet: IRowSet);
Retrieves column information from a supplied IRowSet
- is used e.g. by TOleDBStatement.Execute or to retrieve metadata columns
- raise an exception on error

procedure Prepare(const aSQL: RawUTF8; ExpectResults: Boolean=false); overload;
override;
Prepare an UTF-8 encoded SQL statement
- parameters marked as ? will be bound later, before ExecutePrepared call
- if ExpectResults is TRUE, then Step() and Column*() methods are available to retrieve the data rows
- raise an EOleDBException on any error

procedure ReleaseRows; override;
Clear result rowset when ISQLDBStatement is back in cache

procedure Reset; override;
Reset the previous prepared statement
- this overridden implementation will reset all bindings and the cursor state
- raise an EOleDBException on any error

property AlignDataInternalBuffer: boolean read fAlignBuffer write fAlignBuffer;
If TRUE, the data will be 8 bytes aligned in OleDB internal buffers
- it's recommended by official OleDB documentation for faster process
- is enabled by default, and should not be modified in most cases

property OleDBConnection: TOleDBConnection read fOleDBConnection;
Just map the original Collection into a TOleDBConnection class

property RowBufferSize: integer read fRowBufferSize write SetRowBufferSize;
Size in bytes of the internal OleDB buffer used to fetch rows
- several rows are retrieved at once into the internal buffer
- default value is 16384 bytes, minimal allowed size is 8192 bytes

Types implemented in the SynOleDB unit

TOLeDBBindStatus = ( bsOK, bsBadOrdinal, bsUnsupportedConversion, bsBadBindInfo,
bsBadStorageFlags, bsNoInterface, bsMultipleStorage );
Binding status of a given column
- see http://msdn.microsoft.com/en-us/library/windows/desktop/ms720969 and

TOLeDBMSSQL2008ConnectionProperties = TOleDBMSSQLConnectionProperties;
OleDB connection properties to Microsoft SQL Server 2008, via SQL Server Native Client 10.0 (SQL Server 2008)
- just maps default TOleDBMSSQLConnectionProperties type

TOLeDBStatementParamDynArray = array of TOLeDBStatementParam;
Used to store properties about TOLeDBStatement Parameters
- whole memory block of a TOLeDBStatementParamDynArray will be used as the source Data for the OleDB parameters

TOLeDBStatus = ( stOK, stBadAccessor, stCanNotConvertValue, stIsNull, stTruncated,
stSignMismatch, stDataoverflow, stCanNotCreateValue, stUnavailable,

SynOleDB.pas unit - Rev. 1.18   Page 1472 of 2552

*Indicates whether the data value or some other value, such as a NULL, is to be used as the value of the column or parameter*

### Functions or procedures implemented in the SynOleDb unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoInit</td>
<td>This global procedure should be called for each thread needing to use OLE</td>
<td>1472</td>
</tr>
<tr>
<td>CoUninit</td>
<td>This global procedure should be called at thread termination</td>
<td>1472</td>
</tr>
<tr>
<td>IsJetFile</td>
<td>Check from the file beginning if sounds like a valid Jet / MSAccess file</td>
<td>1472</td>
</tr>
</tbody>
</table>

*procedure CoInit;*

*This global procedure should be called for each thread needing to use OLE*
- it is already called by TOleDbConnection.Create when an OleDb connection is instantiated for a new thread
- every call of CoInit shall be followed by a call to CoUninit
- implementation will maintain some global counting, to call the CoInitialize API only once per thread
- only made public for user convenience, e.g. when using custom COM objects

*procedure CoUninit;*

*This global procedure should be called at thread termination*
- it is already called by TOleDbConnection.Destroy, e.g. when thread associated to an OleDb connection is terminated
- every call of CoInit shall be followed by a call to CoUninit
- only made public for user convenience, e.g. when using custom COM objects

*function IsJetFile(const FileName: TFileName): boolean;*

*Check from the file beginning if sounds like a valid Jet / MSAccess file*
27.29. SynPdf.pas unit

Purpose: PDF file generation
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The SynPdf unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.3.2</td>
<td>A reporting feature, with full preview and export as PDF or TXT files, shall be integrated</td>
<td>2550</td>
</tr>
</tbody>
</table>

Units used in the SynPdf unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
<tr>
<td></td>
<td>- implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynGdiPlus</td>
<td>GDI+ library API access</td>
<td>1350</td>
</tr>
<tr>
<td></td>
<td>- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- make available most useful GDI+ drawing methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- allows Antialiased rending of any EMF file using GDI+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLZ</td>
<td>SynLZ Compression routines</td>
<td>1393</td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynZip</td>
<td>Low-level access to ZLib compression (1.2.5 engine version)</td>
<td>1845</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>
Objects implemented in the **SynPdf** unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPdfInvalidOperation</td>
<td>PDF exception, raised when an invalid operation is triggered</td>
<td>1477</td>
</tr>
<tr>
<td>EPdfInvalidValue</td>
<td>PDF exception, raised when an invalid value is given to a constructor</td>
<td>1477</td>
</tr>
<tr>
<td>TCmapFmt4</td>
<td>Header for the 'cmap' Format 4 table</td>
<td>1477</td>
</tr>
<tr>
<td>TCmapHEAD</td>
<td>The 'head' table contains global information about the font</td>
<td>1477</td>
</tr>
<tr>
<td>TCmapHeader</td>
<td>The 'cmap' table begins with an index containing the table version number followed by the number of encoding tables. The encoding subtables follow.</td>
<td>1477</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TCMapHHEA</td>
<td>Platform identifier Platform-specific encoding identifier Offset of the</td>
<td>1477</td>
</tr>
<tr>
<td></td>
<td>mapping table The <code>hhea</code> table contains information needed to layout fonts</td>
<td></td>
</tr>
<tr>
<td></td>
<td>whose characters are written horizontally, that is, either left to right or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>right to left</td>
<td></td>
</tr>
<tr>
<td>TPdfArray</td>
<td>Used to store an array of PDF objects</td>
<td>1482</td>
</tr>
<tr>
<td>TPdfBinary</td>
<td>Used to handle object which are not defined in this library</td>
<td>1486</td>
</tr>
<tr>
<td>TPdfBoolean</td>
<td>A PDF object, storing a boolean value</td>
<td>1481</td>
</tr>
<tr>
<td>TPdfBox</td>
<td>A PDF coordinates box</td>
<td>1477</td>
</tr>
<tr>
<td>TPdfCanvas</td>
<td>Access to the PDF Canvas, used to draw on the page</td>
<td>1494</td>
</tr>
<tr>
<td>TPdfCatalog</td>
<td>A dictionary wrapper class for the PDF document catalog fields</td>
<td>1502</td>
</tr>
<tr>
<td>TPdfClearText</td>
<td>A PDF object, storing a textual value with no encryption</td>
<td>1482</td>
</tr>
<tr>
<td>TPdfDestination</td>
<td>A destination defines a particular view of a document, consisting of the</td>
<td>1506</td>
</tr>
<tr>
<td></td>
<td>following:</td>
<td></td>
</tr>
<tr>
<td>TPdfDictionary</td>
<td>A PDF Dictionary is used to manage Key / Value pairs</td>
<td>1484</td>
</tr>
<tr>
<td>TPdfDictionaryElement</td>
<td>PDF dictionary element definition</td>
<td>1483</td>
</tr>
<tr>
<td>TPdfDictionaryWrapper</td>
<td>Common ancestor to all dictionary wrapper classes</td>
<td>1502</td>
</tr>
<tr>
<td>TPdfDocument</td>
<td>The main class of the PDF engine, processing the whole PDF document</td>
<td>1487</td>
</tr>
<tr>
<td>TPdfDocumentGDI</td>
<td>Class handling PDF document creation using GDI commands</td>
<td>1508</td>
</tr>
<tr>
<td>TPdfEncryption</td>
<td>Abstract class to handle PDF security</td>
<td>1477</td>
</tr>
<tr>
<td>TPdfEncryptionRC4MD5</td>
<td>Handle PDF security with RC4+MD5 scheme in 40-bit and 128-bit</td>
<td>1478</td>
</tr>
<tr>
<td>TPdfFont</td>
<td>A generic PDF font object</td>
<td>1503</td>
</tr>
<tr>
<td>TPdfFontCIDFontType2</td>
<td>An embedded Composite CIDFontType2</td>
<td>1504</td>
</tr>
<tr>
<td>TPdfFontTrueType</td>
<td>Handle TrueType Font</td>
<td>1505</td>
</tr>
<tr>
<td>TPdfFontType1</td>
<td>An embedded WinAnsi-Encoded standard Type 1 font</td>
<td>1504</td>
</tr>
<tr>
<td>TPdfFontWinAnsi</td>
<td>A generic PDF font object, handling at least WinAnsi encoding</td>
<td>1503</td>
</tr>
<tr>
<td>TPdfForm</td>
<td>Handle any form XObject</td>
<td>1510</td>
</tr>
<tr>
<td>TPdfFormWithCanvas</td>
<td>A form XObject with a Canvas for drawing</td>
<td>1511</td>
</tr>
<tr>
<td>TPdfImage</td>
<td>Generic image object</td>
<td>1510</td>
</tr>
<tr>
<td>TPdfInfo</td>
<td>A dictionary wrapper class for the PDF document information fields</td>
<td>1502</td>
</tr>
<tr>
<td>TPdfName</td>
<td>A PDF object, storing a PDF name</td>
<td>1482</td>
</tr>
<tr>
<td>TPdfNull</td>
<td>A PDF object, storing a NULL value</td>
<td>1481</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TPdfNumber</td>
<td>A PDF object, storing a numerical (integer) value</td>
<td>1481</td>
</tr>
<tr>
<td>TPdfObject</td>
<td>Master class for most PDF objects declaration</td>
<td>1481</td>
</tr>
<tr>
<td>TPdfObjectMgr</td>
<td>Object manager is a virtual class to manage instance of indirect PDF objects</td>
<td>1481</td>
</tr>
<tr>
<td>TPdfObjectStream</td>
<td>Used to handle compressed object stream (in PDF 1.5 format)</td>
<td>1511</td>
</tr>
<tr>
<td>TPdfOptionalContentGroup</td>
<td>Generic PDF Optional Content entry</td>
<td>1487</td>
</tr>
<tr>
<td>TPdfOutlineEntry</td>
<td>An Outline entry in the PDF document</td>
<td>1507</td>
</tr>
<tr>
<td>TPdfOutlineRoot</td>
<td>Root entry for all Outlines of the PDF document</td>
<td>1508</td>
</tr>
<tr>
<td>TPdfOutlines</td>
<td>Generic PDF Outlines entries, stored as a PDF dictionary</td>
<td>1487</td>
</tr>
<tr>
<td>TPdfPage</td>
<td>A PDF page</td>
<td>1493</td>
</tr>
<tr>
<td>TPdfPageGDI</td>
<td>A PDF page, with its corresponding Meta File and Canvas</td>
<td>1508</td>
</tr>
<tr>
<td>TPdfRawText</td>
<td>A PDF object, storing a raw PDF content</td>
<td>1482</td>
</tr>
<tr>
<td>TPdfReal</td>
<td>A PDF object, storing a numerical (floating point) value</td>
<td>1481</td>
</tr>
<tr>
<td>TPdfRect</td>
<td>A PDF coordinates rectangle</td>
<td>1477</td>
</tr>
<tr>
<td>TPdfStream</td>
<td>A temporary memory stream, to be stored into the PDF content</td>
<td>1485</td>
</tr>
<tr>
<td>TPdfText</td>
<td>A PDF object, storing a textual value</td>
<td>1482</td>
</tr>
<tr>
<td>TPdfTextString</td>
<td>A PDF object, storing a textual value</td>
<td>1482</td>
</tr>
<tr>
<td>TPdfTextUTF8</td>
<td>A PDF object, storing a textual value</td>
<td>1482</td>
</tr>
<tr>
<td>TPdfTrailer</td>
<td>The Trailer of the PDF File</td>
<td>1486</td>
</tr>
<tr>
<td>TPdfTTF</td>
<td>Handle Unicode glyph description for a True Type Font</td>
<td>1504</td>
</tr>
<tr>
<td>TPdfVirtualObject</td>
<td>A virtual PDF object, with an associated PDF Object Number</td>
<td>1481</td>
</tr>
<tr>
<td>TPdfWrite</td>
<td>Buffered writer class, specialized for PDF encoding</td>
<td>1478</td>
</tr>
<tr>
<td>TPdfXObject</td>
<td>Any object stored to the PDF file</td>
<td>1487</td>
</tr>
<tr>
<td>TPdfXref</td>
<td>Store the XRef list of the PDF file</td>
<td>1487</td>
</tr>
<tr>
<td>TPdfXrefEntry</td>
<td>Store one entry in the XRef list of the PDF file</td>
<td>1486</td>
</tr>
<tr>
<td>TScriptAnalysis</td>
<td>An Uniscribe script analysis</td>
<td>1511</td>
</tr>
<tr>
<td>TScriptItem</td>
<td>A Uniscribe script item, after analysis of a unicode text</td>
<td>1511</td>
</tr>
<tr>
<td>TScriptProperties</td>
<td>Contains information about Uniscribe special processing for each script</td>
<td>1512</td>
</tr>
<tr>
<td>TScriptState</td>
<td>An UniScribe script state</td>
<td>1511</td>
</tr>
</tbody>
</table>
TCmapHeader = packed record
The 'cmap' table begins with an index containing the table version number followed by the number of encoding tables. The encoding subtables follow.

numberSubtables: word;
Number of encoding subtables

version: word;
Version number (Set to zero)

TCmapHHEA = packed record
Platform identifier Platform-specific encoding identifier Offset of the mapping table The 'hhea' table contains information needed to layout fonts whose characters are written horizontally, that is, either left to right or right to left

TCmapHEAD = packed record
The 'head' table contains global information about the font

TCmapFmt4 = packed record
Header for the 'cmap' Format 4 table
- this is a two-byte encoding format

EPdfInvalidValue = class(Exception)
PDF exception, raised when an invalid value is given to a constructor

EPdfInvalidOperation = class(Exception)
PDF exception, raised when an invalid operation is triggered

TPdfRect = record
A PDF coordinates rectangle

TPdfBox = record
A PDF coordinates box

TPdfEncryption = class(TObject)
Abstract class to handle PDF security

constructor Create(aLevel: TPdfEncryptionLevel; aPermissions: TPdfEncryptionPermissions; const aUserPassword, aOwnerPassword: string); virtual;
Initialize the internal structures with the proper classes
- do not call this method directly, but class function TPdfEncryption.New()
class function New(aLevel: TPdfEncryptionLevel; const aUserPassword, aOwnerPassword: string; aPermissions: TPdfEncryptionPermissions): TPdfEncryption;

Will create the expected TPdfEncryption instance, depending on aLevel
- to be called as parameter of TPdfDocument/TPdfDocumentGDI.Create()
- currently, only elRC4_40 and elRC4_128 levels are implemented
- both passwords are expected to be ASCII-7 characters only
- aUserPassword will be asked at file opening; to be set to '' for not blocking display, but optional permission
- aOwnerPassword shall not be '', and will be used internally to cypher the pdf file content
- aPermissions can be either one of the PDF_PERMISSION_ALL / PDF_PERMISSION_NOMODIF / PDF_PERMISSION_NOPRINT / PDF_PERMISSION_NOCOPY / PDF_PERMISSION_NOCOPYNORPRINT set of options
- typical use may be:
  Doc := TPdfDocument.Create(false,0,false,
      TPdfEncryption.New(elRC4_40,'','',PDF_PERMISSION_NOMODIF));
  Doc := TPdfDocument.Create(false,0,false,
      TPdfEncryption.New(elRC4_128,'','',PDF_PERMISSION_NOCOPYNORPRINT));

procedure AttachDocument(aDoc: TPdfDocument); virtual;
  Prepare a specific document to be encrypted
  - internally used by TPdfDocument.NewDoc method

TPdfEncryptionRC4MD5 = class(TPdfEncryption)
Handle PDF security with RC4+MD5 scheme in 40-bit and 128-bit
- allowed aLevel parameters for Create() are only elRC4_40 and elRC4_128

procedure AttachDocument(aDoc: TPdfDocument); override;
  Prepare a specific document to be encrypted
  - will compute the internal keys

TPdfWrite = class(TObject)
Buffered writer class, specialized for PDF encoding
constructor Create(Destination: TPdfDocument; DestStream: TStream);
  Create the buffered writer, for a specified destination stream
function Add(Value: TSynExtended): TPdfWrite; overload;
  Add a floating point numerical value to the buffer
  - up to 2 decimals are written
function Add(Text: PAnsiChar; Len: integer): TPdfWrite; overload;
  Direct raw write of some data
  - no conversion is made
function Add(const Text: RawByteString): TPdfWrite; overload;
  Direct raw write of some data
  - no conversion is made
function Add(Value, DigitCount: Integer): TPdfWrite; overload;
  Add an integer numerical value to the buffer
  - with a specified fixed number of digits (left filled by '0')

function Add(Value: Integer): TPdfWrite; overload;
  Add an integer numerical value to the buffer

function Add(c: AnsiChar): TPdfWrite; overload;
  Add a character to the buffer

function AddColorStr(Color: TPdfColorRGB): TPdfWrite;
  Add a PDF color, from its TPdfColorRGB RGB value

function AddEscape(Text: PAnsiChar; TextLen: integer): TPdfWrite;
  Add some WinAnsi text as PDF text
  - used by TPdfText object

function AddEscapeContent(const Text: RawByteString): TPdfWrite;
  Add some WinAnsi text as PDF text
  - used by TPdfText object
  - will optionally encrypt the content

function AddEscapeName(Text: PAnsiChar): TPdfWrite;
  Add some PDF /property value

function AddEscapeText(Text: PAnsiChar; Font: TPdfFont): TPdfWrite;
  Add some WinAnsi text as PDF text
  - used by TPdfCanvas.ShowText method for WinAnsi text

function AddGlyphs(Glyphs: PWord; GlyphsCount: integer; Canvas: TPdfCanvas;
  AVisAttrsPtr: Pointer=nil): TPdfWrite;
  Write some Unicode text, encoded as Glyphs indexes, corresponding to the current font

function AddHex(const Bin: PDFString): TPdfWrite;
  Hexadecimal write of some row data
  - row data is written as hexadecimal byte values, one by one

function AddHex4(aWordValue: cardinal): TPdfWrite;
  Add a word value, as Big-Endian 4 hexadecimal characters

function AddIntegerBin(value: integer; bytesize: cardinal): TPdfWrite;
  Add an integer value as binary, specifying a storage size in bytes

function AddIso8601(DateTime: TDateTime): TPdfWrite;
  Add an ISO 8601 encoded date time (e.g. '2010-06-16T15:06:59-07:00')

function AddToUnicodeHex(const Text: PDFString): TPdfWrite;
  Convert some text into unicode characters, then write it as as Big-Endian 4 hexadecimal characters
  - Ansi to Unicode conversion uses the CodePage set by Create() constructor
function AddToUnicodeHexText(const Text: PDFString; NextLine: boolean; Canvas: TPdfCanvas): TPdfWrite;

Convert some text into unicode characters, then write it as PDF Text
- Ansi to Unicode conversion uses the CodePage set by Create() constructor
- use (...) for all WinAnsi characters, or <..hexa..> for Unicode characters
- if NextLine is TRUE, the first written PDF Text command is not Tj but '
- during the text process, corresponding TPdfTrueTypeFont properties are updated (Unicode
version created if necessary, indicate used glyphs for further Font properties writing to the PDF
file content...)
- if the current font is not True Type, all Unicode characters are drawn as '?'

function AddUnicodeHex(PW: PWideChar; WideCharCount: integer): TPdfWrite;
Write some unicode text as as Big-Endian 4 hexadecimal characters

function AddUnicodeHexText(PW: PWideChar; NextLine: boolean; Canvas: TPdfCanvas): TPdfWrite;
Write some Unicode text, as PDF text
- incoming unicode text must end with a #0
- use (...) for all WinAnsi characters, or <..hexa..> for Unicode characters
- if NextLine is TRUE, the first written PDF Text command is not Tj but '
- during the text process, corresponding TPdfTrueTypeFont properties are updated (Unicode
version created if necessary, indicate used glyphs for further Font properties writing to the PDF
file content...)
- if the current font is not True Type, all Unicode characters are drawn as '?'

function AddWithSpace(Value: TSynExtended): TPdfWrite; overload;
Add a floating point numerical value to the buffer
- up to 2 decimals are written, together with a trailing space

function AddWithSpace(Value: Integer): TPdfWrite; overload;
Add an integer numerical value to the buffer
- and append a trailing space

function AddWithSpace(Value: TSynExtended; Decimals: cardinal): TPdfWrite; overload;
Add a floating point numerical value to the buffer
- this version handles a variable number of decimals, together with a trailing space - this is used
by ConcatToCTM e.g. or enhanced precision

function Position: Integer;
Return the current position
- add the current internal buffer stream position to the destination stream position

function ToPDFString: PDFString;
Get the data written to the Writer as a PDFString
- this method could not use Save to flush the data, if all input was inside the internal buffer (save
some CPU and memory): so don't intend the destination stream to be flushed after having called
this method

procedure AddRGB(P: PAnsiChar; PInc, Count: integer);
Add a TBitmap.Scanline[] content into the stream
procedure Save;
    Flush the internal buffer to the destination stream

TPdfObjectMgr = class(TObject)
    Object manager is a virtual class to manage instance of indirect PDF objects

TPdfObject = class(TObject)
    Master class for most PDF objects declaration

constructor Create; virtual;
    Create the PDF object instance

procedure ForceSaveNow;
    Low-level force the object to be saved now
    - you should not use this low-level method, unless you want to force the FSaveAtTheEnd internal flag to be set to force, so that TPdfDocument.SaveToStreamDirectPageFlush would flush the object content

procedure WriteTo(var W: TPdfWrite);
    Write object to specified stream
    - If object is indirect object then write references to stream

procedure WriteValueTo(var W: TPdfWrite);
    Write indirect object to specified stream
    - this method called by parent object

property GenerationNumber: integer read FGenerationNumber;
    The associated PDF Generation Number

property ObjectNumber: integer read FObjectNumber write SetObjectNumber;
    The associated PDF Object Number
    - If you set an object number higher than zero, the object is considered as indirect. Otherwise, the object is considered as direct object.

property ObjectType: TPdfObjectType read FObjectType;
    The corresponding type of this PDF object

TPdfVirtualObject = class(TPdfObject)
    A virtual PDF object, with an associated PDF Object Number

TPdfBoolean = class(TPdfObject)
    A PDF object, storing a boolean value

TPdfNull = class(TPdfObject)
    A PDF object, storing a NULL value

TPdfNumber = class(TPdfObject)
    A PDF object, storing a numerical (integer) value

TPdfReal = class(TPdfObject)
    A PDF object, storing a numerical (floating point) value
**TPdfText** = **class**(TPdfObject)

* A PDF object, storing a textual value
  - the value is specified as a PDFString
  - this object is stored as 'escapedValue'
  - in case of MBCS, conversion is made into Unicode before writing, and stored as '<FEFFHexUnicodeEncodedValue>'

**TPdfTextUTF8** = **class**(TPdfObject)

* A PDF object, storing a textual value
  - the value is specified as an UTF-8 encoded string
  - this object is stored as 'escapedValue'
  - in case characters with ANSI code higher than 8 Bits, conversion is made into Unicode before writing, and '<FEFFHexUnicodeEncodedValue>'

**TPdfTextString** = **class**(TPdfTextUTF8)

* A PDF object, storing a textual value
  - the value is specified as a generic VCL string
  - this object is stored as 'escapedValue'
  - in case characters with ANSI code higher than 8 Bits, conversion is made into Unicode before writing, and '<FEFFHexUnicodeEncodedValue>'

**TPdfRawText** = **class**(TPdfText)

* A PDF object, storing a raw PDF content
  - this object is stored into the PDF stream as the defined Value

**TPdfClearText** = **class**(TPdfText)

* A PDF object, storing a textual value with no encryption
  - the value is specified as a memory buffer
  - this object is stored as 'escapedValue'

**TPdfName** = **class**(TPdfText)

* A PDF object, storing a PDF name
  - this object is stored as '/Value'

**TPdfArray** = **class**(TPdfObject)

* Used to store an array of PDF objects

  **constructor** Create(AObjectMgr: TPdfObjectMgr; AArray: PWordArray; AArrayCount: integer); reintroduce; overload;

  * Create an array of PDF objects, with some specified TPdfNumber values

  **constructor** Create(AObjectMgr: TPdfObjectMgr; const AArray: array of Integer); reintroduce; overload;

  * Create an array of PDF objects, with some specified TPdfNumber values

  **constructor** Create(AObjectMgr: TPdfObjectMgr); reintroduce; overload;

  * Create an array of PDF objects

  **constructor** CreateNames(AObjectMgr: TPdfObjectMgr; const AArray: array of PDFString); reintroduce; overload;

  * Create an array of PDF objects, with some specified TPdfName values
constructor CreateReals(AObjectMgr: TPdfObjectMgr; const AArray: array of double);
reintroduce; overload;

Create an array of PDF objects, with some specified TPdfReal values

destructor Destroy; override;
Release the instance memory, and all embedded objects instances

function AddItem(AItem: TPdfObject): integer;
Add a PDF object to the array
- if AItem already exists, do nothing

function FindName(const AName: PDFString): TPdfName;
Retrieve a TPDFName object stored in the array

function RemoveName(const AName: PDFString): boolean;
Remove a specified TPDFName object stored in the array

procedure InsertItem(Index: Integer; AItem: TPdfObject);
Insert a PDF object to the array
- if AItem already exists, do nothing

property ItemCount: integer read GetItemCount;
Retrieve the array size

property Items[Index: integer]: TPdfObject read GetItems;
Retrieve an object instance, stored in the array

property List: TList read FArray;
Direct access to the internal TList instance
- not to be used normally

property ObjectMgr: TPdfObjectMgr read FObjectMgr;
The associated PDF Object Manager

TPdfDictionaryElement = class(TObject)

PDF dictionary element definition

constructor Create(const AKey: PDFString; AValue: TPdfObject; AInternal: Boolean=false);
Create the corresponding Key / Value pair

destructor Destroy; override;
Release the element instance, and both associated Key and Value

property IsInternal: boolean read FIsInternal;
If this element was created as internal, i.e. not to be saved to the PDF content

property Key: PDFString read GetKey;
The associated Key Name

property Value: TPdfObject read FValue;
The associated Value stored in this element
TPdfDictionary = class(TPdfObject)

A PDF Dictionary is used to manage Key / Value pairs

constructor Create(AObjectMgr: TPdfObjectMgr); reintroduce;
Create the PDF dictionary

destructor Destroy; override;
Release the dictionary instance, and all associated elements

function PdfArrayByName(const AKey: PDFString): TPdfArray;
Fast find an array value by its name

function PdfBooleanByName(const AKey: PDFString): TPdfBoolean;
Fast find a boolean value by its name

function PdfDictionaryByName(const AKey: PDFString): TPdfDictionary;
Fast find a dictionary value by its name

function PdfNameByName(const AKey: PDFString): TPdfName;
Fast find a name value by its name

function PdfNumberByName(const AKey: PDFString): TPdfNumber;
Fast find a numerical (integer) value by its name

function PdfRealByName(const AKey: PDFString): TPdfReal;
Fast find a numerical (floating-point) value by its name

function PdfTextByName(const AKey: PDFString): TPdfText;
Fast find a textual value by its name

function PdfTextStringValueByName(const AKey: PDFString): string;
Fast find a textual value by its name
- return "" if not found, the TPdfTextString.Value otherwise

function PdfTextUTF8ValueByName(const AKey: PDFString): RawUTF8;
Fast find a textual value by its name
- return "" if not found, the TPdfTextUTF8.Value otherwise

function PdfTextValueByName(const AKey: PDFString): PDFString;
Fast find a textual value by its name
- return "" if not found, the TPdfText.Value otherwise

function ValueByName(const AKey: PDFString): TPdfObject;
Fast find a value by its name

procedure AddItem(const AKey: PDFString; AValue: Integer); overload;
Add a specified Key / Value pair (of type TPdfNumber) to the dictionary

procedure AddItem(const AKey, AValue: PDFString); overload;
Add a specified Key / Value pair (of type TPdfName) to the dictionary
procedure AddItem(const AKey: PDFString; AValue: TPdfObject; AInternal: Boolean=false); overload;
   Add a specified Key / Value pair to the dictionary
   - create PdfDictionaryElement with given key and value, and add it to list
   - if the element exists, replace value of element by given value
   - internal items are local to the framework, and not to be saved to the PDF content

procedure AddItemText(const AKey, AValue: PDFString); overload;
   Add a specified Key / Value pair (of type TPdfText) to the dictionary

procedure AddItemTextString(const AKey: PDFString; const AValue: string); overload;
   Add a specified Key / Value pair (of type TPdfTextUTF8) to the dictionary
   - the value is a generic VCL string: it will be written as Unicode hexadecimal to the PDF stream, if necessary

procedure AddItemTextUTF8(const AKey: PDFString; const AValue: RawUTF8); overload;
   Add a specified Key / Value pair (of type TPdfTextUTF8) to the dictionary
   - the value can be any UTF-8 encoded text: it will be written as Unicode hexadecimal to the PDF stream, if necessary

procedure RemoveItem(const AKey: PDFString);
   Remove the element specified by its Key from the dictionary
   - if the element does not exist, do nothing

property ItemCount: integer read GetItemCount;
   Retrieve the dictionary element count

property Items[Index: integer]: TPdfDictionaryElement read GetItems;
   Retrieve any dictionary element

property List: TList read FArray;
   Direct access to the internal TList instance
   - not to be used normally

property ObjectMgr: TPdfObjectMgr read FObjectMgr;
   Retrieve the associated Object Manager

property TypeOf: PDFString read getTypeOf;
   Retrieve the type of the pdfdictionary object, i.e. the 'Type' property name

TPdfStream = class(TPdfObject)
   A temporary memory stream, to be stored into the PDF content
   - typically used for the page content
   - can be compressed, if the FlateDecode filter is set

constructor Create(ADoc: TPdfDocument; DontAddToFXref: boolean=false);
   reintroduce;
      Create the temporary memory stream
      - an optional DontAddToFXref is available, if you don't want to add this object to the main XRef list of the PDF file

destructor Destroy; override;
   Release the memory stream
```pascal
property Attributes: TPdfDictionary read FAttributes;
  ― Retrieve the associated attributes, e.g. the stream Length

property Filter: PDFString read FFilter write FFilter;
  ― Retrieve the associated filter name

property Writer: TPdfWrite read FWriter;
  ― Retrieve the associated buffered writer
  - use this TPdfWrite instance to write some data into the stream

TPdfBinary = class(TPdfObject)
  Used to handle object which are not defined in this library

constructor Create; override;
  Create the instance, i.e. its associated stream

destructor Destroy; override;
  Release the instance

property Stream: TMemoryStream read FStream;
  The associated memory stream, used to store the corresponding data
  - the content of this stream will be written to the resulting

TPdfTrailer = class(TObject)
  The Trailer of the PDF File

TPdfXrefEntry = class(TObject)
  Store one entry in the XRef list of the PDF file

constructor Create(AValue: TPdfObject);
  Create the entry, with the specified value
  - if the value is nil (e.g. root entry), the type is 'f' (PDF_FREE_ENTRY), otherwise the entry type is 'n' (PDF_IN_USE_ENTRY)

destructor Destroy; override;
  Release the memory, and the associated value, if any

procedure SaveToPdfWrite(var W: TPdfWrite);
  Write the XRef list entry

property ByteOffset: integer read FByteOffset;
  The position (in bytes) in the PDF file content stream
  - to be ignored if ObjectStreamIndex>=0

property EntryType: PDFString read FEntryType write FEntryType;
  Return either 'f' (PDF_FREE_ENTRY), either 'n' (PDF_IN_USE_ENTRY)

property GenerationNumber: integer read FGenerationNumber write FGenerationNumber;
  The associated Generation Number
  - mostly 0, or 65535 (PDF_MAX_GENERATION_NUM) for the root 'f' entry
```
**property** ObjectStreamIndex: Integer read FObjectStreamIndex;

_The index of this object in the global compressed /ObjStm object stream_
- equals -1 by default, i.e. if stored within the main file content stream

**property** Value: TPdfObject read FValue;

_The associated PDF object_

---

**TPdfXref** = **class**(TPdfObjectMgr)

_Store the XRef list of the PDF file_

**constructor** Create;

_Initialize the XRef object list_
- create first a void 'f' (PDF_FREE_ENTRY) as root

**destructor** Destroy; **override**;

_Release instance memory and all associated XRef objects_

**function** GetObject(ObjectID: integer): TPdfObject; **override**;

_Retrieve an object from its object ID_

**procedure** AddObject(AObject: TPdfObject); **override**;

_Register object to the xref table, and set corresponding object ID_

**property** ItemCount: integer read GetItemCount;

_Retrieve the XRef object count_

**property** Items[ObjectID: integer]: TPdfXrefEntry read GetItem;

_Retrieve a XRef object instance, from its object ID_

---

**TPdfXObject** = **class**(TPdfStream)

_Any object stored to the PDF file_
- these objects are the main unit of the PDF file content
- these objects are written in the PDF file, followed by a "xref" table

**TPdfOutlines** = **class**(TPdfDictionary)

_Generic PDF Outlines entries, stored as a PDF dictionary_

**TPdfOptionalContentGroup** = **class**(TPdfDictionary)

_Generic PDF Optional Content entry_

---

**TPdfDocument** = **class**(TObject)

_The main class of the PDF engine, processing the whole PDF document_

_Used for DI-2.3.2 (page 2550)._
constructor Create(AUseOutlines: Boolean=false; ACodePage: integer=0; APDFA1: boolean=false ; AEncryption: TPdfEncryption=nil); reintroduce;

Create the PDF document instance, with a Canvas and a default A4 paper size
- the current charset and code page are retrieved from the SysLocale value, so the PDF engine is MBCS ready
- note that only Win-Ansi encoding allows use of embedded standard fonts
- you can specify a Code Page to be used for the PDFString encoding; by default (ACodePage left to 0), the current system code page is used
- you can create a PDF/A-1 compliant document by setting APDFA1 to true
- you can set an encryption instance, by using TPdfEncryption.New()

destructor Destroy; override;
Release the PDF document instance

function AddPage: TPdfPage; virtual;
Add a Page to the current PDF document

function AddXObject(const AName: PDFString; AXObject: TPdfXObject): integer;
Add then register an object (typically a TPdfImage) to the PDF document
- returns the internal index as added in FXObjectList[]

function CreateAnnotation(AType: TPdfAnnotationSubType; const ARect: TPdfRect; BorderStyle: TPdfAnnotationBorder=abSolid; BorderWidth: integer=1): TPdfDictionary;
Wrapper to create an annotation
- the annotation is set to a specified position of the current page

function CreateDestination: TPdfDestination;
Create a Destination
- the current PDF Canvas page is associated with this destination object

function CreateHyperLink(const ARect: TPdfRect; const url: RawUTF8; BorderStyle: TPdfAnnotationBorder=abSolid; BorderWidth: integer=0): TPdfDictionary;
Wrapper to create a hyper-link, with a specific URL value

function CreateLink(const ARect: TPdfRect; const aBookmarkName: RawUTF8; BorderStyle: TPdfAnnotationBorder=abSolid; BorderWidth: integer=1): TPdfDictionary;
Wrapper to create a Link annotation, specified by a bookmark
- the link is set to a specified rectangular position of the current page
- if the bookmark name is not existing (i.e. if it no such name has been defined yet via the CreateBookMark method), it's added to the internal fMissingBookmarks list, and will be linked at CreateBookMark method call

function CreateOptionalContentGroup(ParentContentGroup: TPdfOptionalContentGroup; const Title: string; Visible: Boolean=true): TPdfOptionalContentGroup;
Create a new optional content group (layer)
- returns a TPdfOptionalContentGroup needed for TPDFCanvas.BeginMarkedContent
- if ParentContentGroup is not nil, the new content group is a subgroup to ParentContentGroup
- Title is the string shown in the PDF Viewer
- Visible controls the initial state of the content group
function CreateOrGetImage(B: TBitmap; DrawAt: PPdfBox=nil; ClipRc: PPdfBox=nil): PDFString;

Create an image from a supplied bitmap
- returns the internal XObject name of the resulting TPDFImage
- if you specify a PPdfBox to draw the image at the given position/size
- if the same bitmap content is sent more than once, the TPDFImage will be reused (it will therefore spare resulting pdf file space) - if the ForceNoBitmapReuse is FALSE
- if ForceCompression property is set, the picture will be stored as a JPEG
- you can specify a clipping rectangle region as ClipRc parameter

function CreateOutline(const Title: string; Level: integer; TopPosition: Single): TPdfOutlineEntry;

Create an Outline entry at a specified position of the current page
- the outline tree is created from the specified numerical level (0=root), just after the item added via the previous CreateOutline call
- the title is a generic VCL string, to handle fully Unicode support

function CreatePages(Parent: TPdfDictionary): TPdfDictionary;

Create a Pages object
- Pages objects can be nested, to save memory used by the Viewer
- only necessary if you have more than 8000 pages (this method is called by TPdfDocument.NewDoc, so you shouldn't have to use it)

function GetXObject(const AName: PDFString): TPdfXObject;

Retrieve a XObject from its name
- this method will handle also the Virtual Objects

function GetXObjectImageName(const Hash: THash128Rec; Width, Height: Integer): PDFString;

Retrieve a XObject TPdfImage index from its picture attributes
- returns '' if this image is not already there
- uses 4 hash codes, created with 4 diverse seeds, in order to avoid false positives

function GetXObjectIndex(const AName: PDFString): integer;

Retrieve a XObject index from its name
- this method won't handle the Virtual Objects

function RegisterXObject(AObject: TPdfXObject; const AName: PDFString): integer;

Register an object (typically a TPdfImage) to the PDF document
- returns the internal index as added in FXObjectList[]

function SaveToFile(const aFileName: TFileName): boolean;

Save the PDF file content into a specified file
- return FALSE on any writing error (e.g. if the file is opened in the Acrobat Reader)

procedure CreateBookMark(TopPosition: Single; const aBookmarkName: RawUTF8);

Create an internal bookmark entry at a specified position of the current page
- the current PDF Canvas page is associated with the destination object
- a dXyz destination with the corresponding TopPosition Y value is defined
- the associated bookmark name must be unique, otherwise an exception is raised
procedure CreateOptionalContentRadioGroup(const ContentGroups: array of TPdfOptionalContentGroup);

Create a Radio Optional ContentGroup
- ContentGroups is a array of TPdfOptionalContentGroups which should behave like radiosubuttons, i.e. only one active at a time
- visibility must be set with CreateOptionalContentGroup, only one group should be visible

procedure NewDoc;

Create a new document
- this method is called first, by the Create constructor
- you can call it multiple times if you want to reset the whole document content

procedure SaveToStream(AStream: TStream; ForceModDate: TDateTime=0); virtual;

Save the PDF file content into a specified Stream

procedure SaveToStreamDirectBegin(AStream: TStream; ForceModDate: TDateTime=0);

Prepare to save the PDF file content into a specified Stream
- is called by SaveToStream() method
- you can then append other individual pages with SaveToStreamCurrentPage to avoid most resource usage (e.g. for report creation)
- shall be finished by a SaveToStreamDirectEnd call

procedure SaveToStreamDirectEnd;

Prepare to save the PDF file content into a specified Stream
- shall be made once after a SaveToStreamDirectBegin() call
- is called by SaveToStream() method

procedure SaveToStreamDirectPageFlush(FlushCurrentPageNow: boolean=false); virtual;

Save the current page content to the PDF file
- shall be made one or several times after a SaveToStreamDirectBegin() call and before a final SaveToStreamDirectEnd call
- see TPdfDocumentGDI.SaveToStream() in this unit, and TGDIPages.ExportPDFStream() in mORMotReport.pas for real use cases
- you can set FlushCurrentPageNow=true to force the current page to be part of the flushed content

property Canvas: TPdfCanvas read fCanvas;

Retrieve the current PDF Canvas, associated to the current page

property CharSet: integer read FCharSet;

The current CharSet used for this PDF Document

property CodePage: cardinal read FCodePage;

The current Code Page encoding used for this PDF Document

property CompressionMethod: TPdfCompressionMethod read FCompressionMethod write FCompressionMethod;

The compression method used for page content storage
- is set by default to cmFlateDecode when the class instance is created

property DefaultPageHeight: cardinal read FDefaultPageHeight write SetDefaultPageHeight;

The default page height, used for new every page creation (i.e. AddPage method call)
property DefaultPageLandscape: boolean read GetDefaultPageLandscape write SetDefaultPageLandscape;

The default page orientation
- a call to this property will swap default page width and height if the orientation is not correct

property DefaultPageSize: card read FDefaultPageSize write SetDefaultPageSize;

The default page size, used for new every page creation (i.e. AddPage method call)
- a write to this property this will reset the default page orientation to Portrait: you must explicitly set DefaultPageLandscape to true, if needed

property EmbeddedTTF: boolean read fEmbeddedTTF write fEmbeddedTTF;

If set to TRUE, the used True Type fonts will be embedded to the PDF content
- not set by default, to save disk space and produce tiny PDF

property EmbeddedTTFIgnore: TRawUTF8List read GetEmbeddedTTFIgnore;

You can add some font names to this list, if you want these fonts NEVER to be embedded to the PDF file, even if the EmbeddedTTF property is set
- if you want to ignore all standard windows fonts, use:
  EmbeddedTTFIgnore.Text := MSWINDOWS_DEFAULT_FONTS;

property EmbeddedWholeTTF: boolean read fEmbeddedWholeTTF write fEmbeddedWholeTTF;

If set to TRUE, the embedded True Type fonts will be totally Embedded
- by default, is set to FALSE, meaning that a subset of the TTF font is stored into the PDF file, i.e. only the used glyphs are stored
- this option is only available if running on Windows XP or later

property FontFallBackName: string read GetFontFallBackName write SetFontFallBackName;

Set the font name to be used for missing characters
- used only if UseFontFallBack is TRUE
- default value is 'Arial Unicode MS', if existing

property ForceJPEGCompression: integer read fForceJPEGCompression write fForceJPEGCompression;

This property can force saving all canvas bitmaps images as JPEG
- handle bitmaps added by VCLCanvas/TMetaFile and bitmaps added as TPdfImage
- by default, this property is set to 0 by the constructor of this class, meaning that the JPEG compression is not forced, and the engine will use the native resolution of the bitmap - in this case, the resulting PDF file content will be bigger in size (e.g. use this for printing)
- 60 is the preferred way e.g. for publishing PDF over the internet
- 80/90 is a good ratio if you want to have a nice PDF to see on screen
- of course, this doesn't affect vectorial (i.e. emf) pictures
property ForceNoBitmapReuse: boolean read fForceNoBitmapReuse write fForceNoBitmapReuse;

This property can force all canvas bitmaps to be stored directly
- by default, the library will try to match an existing same bitmap content, and reuse the existing pdf object - you can set this property for a faster process, if you do not want to use this feature

property GeneratePDF15File: boolean read GetGeneratePDF15File write SetGeneratePDF15File;

Set to TRUE to force PDF 1.5 format, which may produce smaller files

property Info: TPdfInfo read GetInfo;

Retrieve the PDF information, associated to the PDF document

property OutlineRoot: TPdfOutlineRoot read GetOutlineRoot;

Retrieve the PDF Outline, associated to the PDF document
- UseOutlines must be set to TRUE before any use of the OutlineRoot property

property PDFA1: boolean read fPDFA1 write SetPDFA1;

Is TRUE if the file was created in order to be PDF/A-1 compliant
- set APDFA1 parameter to true for Create constructor in order to use it
- warning: setting a value to this property after creation will call the NewDoc method, therefore will erase all previous content and pages (including Info properties)

property RawPages: TList read fRawPages;

Direct read-only access to all corresponding TPdfPage
- can be useful in inherited classe

property Root: TPdfCatalog read fRoot;

Retrieve the PDF Document Catalog, as root of the document’s object hierarchy

property ScreenLogPixels: Integer read FScreenLogPixels write FScreenLogPixels;

The resolution used for pixel to PDF coordinates conversion
- by default, contains the Number of pixels per logical inch along the screen width
- you can override this value if you really need additional resolution for your bitmaps and such - this is useful only with TPdfDocumentGDI and its associated TCanvas: all TPdfDocument native TPdfCanvas methods use the native resolution of the PDF, i.e. more than 7200 DPI (since we write coordinates with 2 decimals per point - which is 1/72 inch)

property StandardFontsReplace: boolean read FStandardFontsReplace write SetStandardFontsReplace;

Set if the PDF engine must use standard fonts substitution
- if TRUE, 'Arial', 'Times New Roman' and 'Courier New' will be replaced by the corresponding internal Type 1 fonts, defined in the Reader
- only works with current ANSI_CHARSET, i.e. if you want to display some other unicode characters, don't enable this property: all non WinAnsi glyphs would be replaced by a '?' sign
- default value is false (i.e. not embedded standard font)
property UseFontFallBack: boolean read fUseFontFallBack write fUseFontFallBack;

Used to define if the PDF document will handle “font fallback” for characters not existing in the current font: it will avoid rendering block/square symbols instead of the correct characters (e.g. for Chinese text)
- will use the font specified by FontFallBackName property to add any Unicode glyph not existing in the currently selected font
- default value is TRUE

property UseOptionalContent: boolean read FUseOptionalContent write SetUseOptionalContent;

Used to define if the PDF document will use optional content (layers)
- will also force PDF 1.5 as minimal file format
- must be set to TRUE before calling NewDoc
- warning: setting a value to this property after creation will call the NewDoc method, therefore will erase all previous content and pages (including Info properties)

property UseOutlines: boolean read FUseOutlines write FUseOutlines;

Used to define if the PDF document will use outlines
- must be set to TRUE before any use of the OutlineRoot property

property UseUniscribe: boolean read fUseUniscribe write fUseUniscribe;

Set if the PDF engine must use the Windows Uniscribe API to render Ordering and/or Shaping of the text
- useful for Hebrew, Arabic and some Asiatic languages handling
- set to FALSE by default, for faster content generation
- you can set this property temporary to TRUE, when using the Canvas property, but this property must be set appropriately before the content generation if you use any TPdfDocumentGdi.VCLCanvas text output with such scripting (since the PDF rendering is done once just before the saving, e.g. before SaveToFile() or SaveToStream() methods calls)
- the PDF engine don't handle Font Fallback yet: the font you use must contain ALL glyphs necessary for the supplied unicode text - squares or blanks will be drawn for any missing glyph/character

TPdfPage = class(TPdfDictionary)

A PDF page

constructor Create(ADoc: TPdfDocument); reintroduce; virtual;

Create the page with its internal VCL Canvas

function MeasureText(const Text: PDFString; Width: Single): integer;

Calculate the number of chars which can be displayed in the specified width, according to current attributes
- this function is compatible with MBCS strings, and returns the index in Text, not the glyphs index

function TextWidth(const Text: PDFString): Single;

Calculate width of specified text according to current attributes
- this function is compatible with MBCS strings

property CharSpace: Single read fCharSpace write SetCharSpace;

Retrieve or set the Char Space attribute
property Font: TPdfFont read FFont write FFont;
Retrieve the current used font
- for TPdfFontTrueType, this points not always to the WinAnsi version of the Font, but can also
point to the Unicode Version, if the last drawn character by ShowText() was unicode - see
TPdfWrite.AddUnicodeHexText

property FontSize: Single read FFontSize write SetFontSize;
Retrieve or set the font Size attribute

property HorizontalScaling: Single read FHorizontalScaling write SetHorizontalScaling;
Retrieve or set the Horizontal Scaling attribute

property Leading: Single read FLeading write SetLeading;
Retrieve or set the text Leading attribute

property PageHeight: integer read GetPageHeight write SetPageHeight;
Retrieve or set the current page height

property PageLandscape: Boolean read GetPageLandscape write SetPageLandscape;
Retrieve or set the paper orientation

property PageWidth: integer read GetPageWidth write SetPageWidth;
Retrieve or set the current page width

property WordSpace: Single read FWordSpace write SetWordSpace;
Retrieve or set the word Space attribute

TPdfCanvas = class(TObject)
Access to the PDF Canvas, used to draw on the page
Used for DI-2.3.2 (page 2550).

constructor Create(APdfDoc: TPdfDocument);
Create the PDF canvas instance

function GetNextWord(const S: PDFString; var Index: integer): PDFString;
Get the index of the next word in the supplied text
- this function is compatible with MBCS strings, and returns the index in Text, not the glyphs
index

function MeasureText(const Text: PDFString; AWidth: Single): integer;
Calculate the number of chars which can be displayed in the specified width, according to current
attributes
- this function is compatible with MBCS strings, and returns the index in Text, not the glyphs
index
- note: this method only work with embedded fonts by now, not true type fonts (because text
width measuring is not yet implemented for them)
function SetFont(const AName: RawUTF8; ASize: Single; AStyle: TPdfFontStyles; ACharSet: integer=-1; AForceTTF: integer=-1; AIsFixedWidth: boolean=false): TPdfFont; overload;

*Set the current font for the PDF Canvas*
- expect the font name to be either a standard embedded font ('Helvetica','Courier','Times') or its Windows equivalency (i.e. 'Arial','Courier New','Times New Roman'), either a UTF-8 encoded True Type font name available on the system
- if no CharSet is specified (i.e. if it remains -1), the current document CharSet parameter is used

function SetFont(ADC: HDC; const ALogFont: TLogFontW; ASize: single): TPdfFont; overload;

*Set the current font for the PDF Canvas*
- this method use the Win32 structure that defines the characteristics of the logical font

function TextWidth(const Text: PDFString): Single;

*Calculate width of specified text according to current Canvas attributes*
- works with MBCS strings

function UnicodeTextWidth(PW: PWideChar): Single;

*Calculate width of specified text according to current Canvas attributes*
- this function compute the raw width of the specified text, and won't use HorizontalScaling, CharSpace nor WordSpace in its calculation

procedure BeginMarkedContent(Group: TPdfOptionalContentGroup);

*Starts optional content (layer)*
- Group must be registered with TPdfDocument.CreateOptionalContentGroup
- each BeginMarkedContent must have a corresponding EndMarkedContent
- nested BeginMarkedContent/EndMarkedContent are possible

procedure BeginText;

*Begin a text object*
- Text objects cannot be nested

procedure Clip;

*Nonzero winding clipping path set*
- Modify the current clipping path by intersecting it with the current path, using the nonzero winding number rule to determine which regions lie inside the clipping path
- The graphics state contains a clipping path that limits the regions of the page affected by painting operators. The closed subpaths of this path define the area that can be painted. Marks falling inside this area will be applied to the page; those falling outside it will not. (Precisely what is considered to be inside a path is discussed under "Filling", above.)
- The initial clipping path includes the entire page. Both clipping path methods (Clip and EoClip) may appear after the last path construction operator and before the path-painting operator that terminates a path object. Although the clipping path operator appears before the painting operator, it does not alter the clipping path at the point where it appears. Rather, it modifies the effect of the succeeding painting operator. After the path has been painted, the clipping path in the graphics state is set to the intersection of the current clipping path and the newly constructed path.
procedure Closepath;

Close the current subpath by appending a straight line segment from the current point to the
starting point of the subpath
- This operator terminates the current subpath; appending another segment to the current path
will begin a new subpath, even if the new segment begins at the endpoint reached by the h
operation
- If the current subpath is already closed or the current path is empty, it does nothing

procedure ClosepathEofillStroke;

Close, fill, and then stroke the path, using the even-odd rule to determine the region to fill
- This operator has the same effect as the sequence Closepath; EofillStroke;

procedure ClosepathFillStroke;

Close, fill, and then stroke the path, using the nonzero winding number rule to determine the
region to fill
- This operator has the same effect as the sequence ClosePath; FillStroke;

procedure ClosePathStroke;

Close and stroke the path
- This operator has the same effect as the sequence ClosePath; Stroke;

procedure ConcatToCTM(a, b, c, d, e, f: Single; Decimals: Cardinal=6);

Modify the CTM by concatenating the specified matrix
- The current transformation matrix (CTM) maps positions from user coordinates to device
coordinates
- This matrix is modified by each application of the ConcatToCTM method
- CTM Initial value is a matrix that transforms default user coordinates to device coordinates
- since floating-point precision does make sense for a transformation matrix, we added a custom
decimal number parameter here

procedure CurveToC(x1, y1, x2, y2, x3, y3: Single);

Append a cubic Bezier curve to the current path
- The curve extends from the current point to the point (x3, y3), using (x1, y1) and (x2, y2) as the
Bezier control points
- The new current point is (x3, y3)

procedure CurveToV(x2, y2, x3, y3: Single);

Append a cubic Bezier curve to the current path
- The curve extends from the current point to the point (x3, y3), using the current point and (x2,
y2) as the Bezier control points
- The new current point is (x3, y3)

procedure CurveToY(x1, y1, x3, y3: Single);

Append a cubic Bezier curve to the current path
- The curve extends from the current point to the point (x3, y3), using (x1, y1) and (x3, y3) as the
Bezier control points
- The new current point is (x3, y3)

procedure DrawXObject(X, Y, AWidth, AHeight: Single; const AXObjectName: PDFString);

Draw the specified object (typically an image) with stretching
procedure DrawXObjectEx(X, Y, AWidth, AHeight: Single; ClipX, ClipY, ClipWidth, ClipHeight: Single; const AXObjectName: PDFString);
    Draw the specified object (typically an image) with stretching and clipping

procedure Ellipse(x, y, width, height: Single);
    Draw an ellipse
    - use Bezier curves internally to draw the ellipse

procedure EndMarkedContent;
    Ends optional content (layer)

procedure EndText;
    End a text object, discarding the text matrix

procedure EoClip;
    Even-Odd winding clipping path set
    - Modify the current clipping path by intersecting it with the current path, using the even-odd rule to determine which regions lie inside the clipping path

procedure EoFill;
    Fill the path, using the even-odd rule to determine the region to fill

procedure EoFillStroke;
    Fill and then stroke the path, using the even-odd rule to determine the region to fill
    - This operator produces the same result as FillStroke, except that the path is filled as if with EoFill instead of Fill

procedure ExecuteXObject(const xObject: PDFString);
    Paint the specifiedXObject

procedure Fill;
    Fill the path, using the nonzero winding number rule to determine the region to fill

procedure FillStroke;
    Fill and then stroke the path, using the nonzero winding number rule to determine the region to fill
    - This produces the same result as constructing two identical path objects, painting the first with Fill and the second with Stroke. Note, however, that the filling and stroking portions of the operation consult different values of several graphics state parameters, such as the color

procedure GRestore;
    Restores the entire graphics state to its former value by popping it from the stack

procedure GSave;
    Pushes a copy of the entire graphics state onto the stack

procedure LineTo(x, y: Single);
    Append a straight line segment from the current point to the point (x, y).
    - The new current point is (x, y)

procedure MoveTextPoint(tx, ty: Single);
    Move to the start of the next line, offset from the start of the current line by (tx, ty)
    - tx and ty are numbers expressed in unscaled text space units
procedure MoveTo(x, y: Single);

*Change the current coordinates position*
- Begin a new subpath by moving the current point to coordinates (x, y), omitting any connecting line segment. If the previous path construction operator in the current path was also MoveTo(), the new MoveTo() overrides it; no vestige of the previous MoveTo() call remains in the path.

procedure MoveToNextLine;

*Move to the start of the next line*

procedure MultilineTextRect(ARect: TPdfRect; const Text: PDFString; WordWrap: boolean);

*Show the text in the specified rectangle and alignment*
- text can be multiline, separated by CR + LF (i.e. #13#10)
- text can optionally word wrap
- note: this method only works with embedded fonts by now, not true type fonts (because it uses text width measuring)

procedure NewPath;

*End the path object without filling or stroking it*
- This operator is a "path-painting no-op", used primarily for the side effect of changing the clipping path

procedure Rectangle(x, y, width, height: Single);

*Append a rectangle to the current path as a complete subpath, with lower-left corner (x, y) and dimensions width and height in user space*

procedure RenderMetaFile(MF: TMetaFile; ScaleX: Single=1.0; ScaleY: Single=0.0; XOff: single=0.0; YOff: single=0.0; TextPositioning: TPdfCanvasRenderMetaFileTextPositioning=tpSetTextJustification; KerningHScaleBottom: single=99.0; KerningHScaleTop: single=101.0; TextClipping: TPdfCanvasRenderMetaFileTextClipping=tcAlwaysClip);

*Draw a metafile content into the PDF page*
- not 100% of content is handled yet, but most common are (even metafiles embedded inside metafiles)
- UseSetTextJustification is to be set to true to ensure better rendering if the EMF content used SetTextJustification() API call to justify text
- KerningHScaleBottom/KerningHScaleTop are limits below which and over which Font Kerning is transformed into PDF Horizontal Scaling commands
- TextClipping can be set to fix some issues e.g. when using Wine

*Used for DI-2.3.2 (page 2550).*

procedure RoundRect(x1,y1,x2,y2,cx, cy: Single);

*Draw a rounded rectangle*
- use Bezier curves internally to draw the rounded rectangle

procedure SetCharSpace(charSpace: Single);

*Set the character spacing*
- CharSpace is a number expressed in unscaled text space units.
- Character spacing is used by the ShowText and ShowTextNextLine methods
- Default value is 0
procedure SetCMYKFillColor(C, M, Y, K: integer);

*Set the color space to a CMYK percent value*
- this method set the color to use for nonstroking operations

procedure SetCMYKStrokeColor(C, M, Y, K: integer);

*Set the color space to a CMYK value*
- this method set the color to use for stroking operations

procedure SetDash(const aarray: array of integer; phase: integer=0);

*Set the line dash pattern in the graphics state*
- The line dash pattern controls the pattern of dashes and gaps used to stroke paths. It is specified by a dash array and a dash phase. The dash array's elements are numbers that specify the lengths of alternating dashes and gaps; the dash phase specifies the distance into the dash pattern at which to start the dash. The elements of both the dash array and the dash phase are expressed in user space units. Before beginning to stroke a path, the dash array is cycled through, adding up the lengths of dashes and gaps. When the accumulated length equals the value specified by the dash phase, stroking of the path begins, using the dash array cyclically from that point onward.

procedure SetFlat(flatness: Byte);

*Set the flatness tolerance in the graphics state*
- see Section 6.5.1, "Flatness Tolerance" of the PDF 1.3 reference: The flatness tolerance controls the maximum permitted distance in device pixels between the mathematically correct path and an approximation constructed from straight line segments.
- Flatness is a number in the range 0 to 100; a value of 0 specifies the output device's default flatness tolerance

procedure SetFontAndSize(const fontshortcut: PDFString; size: Single);

*Set the font, Tf, to the font size, Tfs, to size.*
- font is the name of a font resource in the Font subdictionary of the current resource dictionary (e.g. 'F0')
- size is a number representing a scale factor
- There is no default value for either font or size; they must be specified using this method before any text is shown

procedure SetHorizontalScaling(hScaling: Single);

*Set the horizontal scaling to (scale/100)*
- hScaling is a number specifying the percentage of the normal width
- Default value is 100 (e.g. normal width)

procedure SetLeading(leading: Single);

*Set the text leading, Ti, to the specified leading value*
- leading which is a number expressed in unscaled text space units; it specifies the vertical distance between the baselines of adjacent lines of text
- Text leading is used only by the MoveToNextLine and ShowTextNextLine methods
- you can force the next line to be just below the current one by calling:
  SetLeading(Attributes.FontSize);
- Default value is 0
procedure SetLineCap(linecap: TLineCapStyle);
  
  *Set the line cap style in the graphics state*
  - The line cap style specifies the shape to be used at the ends of open subpaths (and dashes, if any) when they are stroked

procedure SetLineJoin(linejoin: TLineJoinStyle);

  *Set the line join style in the graphics state*
  - The line join style specifies the shape to be used at the corners of paths that are stroked

procedure SetLineWidth(linewidth: Single);

  *Set the line width in the graphics state*
  - The line width parameter specifies the thickness of the line used to stroke a path. It is a nonnegative number expressed in user space units; stroking a path entails painting all points whose perpendicular distance from the path in user space is less than or equal to half the line width. The effect produced in device space depends on the current transformation matrix (CTM) in effect at the time the path is stroked. If the CTM specifies scaling by different factors in the x and y dimensions, the thickness of stroked lines in device space will vary according to their orientation. The actual line width achieved can differ from the requested width by as much as 2 device pixels, depending on the positions of lines with respect to the pixel grid.

procedure SetMiterLimit(miterlimit: Single);

  *Set the miter limit in the graphics state*
  - When two line segments meet at a sharp angle and mitered joins have been specified as the line join style, it is possible for the miter to extend far beyond the thickness of the line stroking the path. The miter limit imposes a maximum on the ratio of the miter length to the line width. When the limit is exceeded, the join is converted from a miter to a bevel

procedure SetPage(APage: TPdfPage); virtual;
  
  *Assign the canvas to the specified page*

procedure SetPDFFont(AFont: TPdfFont; ASize: Single);

  *Set the current font for the PDF Canvas*

procedure SetRGBFillColor(Value: TPdfColor);

  *Set the color space to a Device-dependent RGB value*
  - this method set the color to use for nonstroking operations

procedure SetRGBStrokeColor(Value: TPdfColor);

  *Set the color space to a Device-dependent RGB value*
  - this method set the color to use for stroking operations

procedure SetTextMatrix(a, b, c, d, x, y: Single);

  *Set the Text Matrix to a,b,c,d and the text line Matrix x,y*

procedure SetTextRenderingMode(mode: TTextRenderingMode);

  *Set the text rendering mode*
  - the text rendering mode determines whether text is stroked, filled, or used as a clipping path
procedure SetTextRise(rise: word);
   Set the text rise, \textit{Rise}, to the specified value
   - rise is a number expressed in unscaled text space units, which specifies the distance, in 
     unscaled text space units, to move the baseline up or down from its default location. Positive 
     values of text rise move the baseline up. Adjustments to the baseline are useful for drawing 
     superscripts or subscripts. The default location of the baseline can be restored by setting the 
     text rise to 0.
   - Default value is 0

procedure SetWordSpace(wordSpace: Single);
   Set the word spacing
   - WordSpace is a number expressed in unscaled text space units
   - word spacing is used by the ShowText and ShowTextNextLine methods
   - Default value is 0

procedure ShowGlyph(PW: PWord; Count: integer);
   Show an Unicode Text string, encoded as Glyphs or the current font
   - PW must follow the \texttt{ETO\_GLYPH\_INDEX} layout, i.e. refers to an array as returned from the 
     GetCharacterPlacement: all glyph indexes are 16-bit values

procedure ShowText(const text: PDFString; NextLine: boolean=false); overload;
   Show a text string
   - text is expected to be Ansi-Encoded, in the current CharSet; if some Unicode or MBCS 
     conversion is necessary, it will be notified to the corresponding
   - if NextLine is TRUE, moves to the next line and show a text string; in this case, method as the 
     same effect as MoveToNextLine; ShowText(s);

procedure ShowText(PW: PWideChar; NextLine: boolean=false); overload;
   Show an Unicode Text string
   - if NextLine is TRUE, moves to the next line and show a text string; in this case, method as the 
     same effect as MoveToNextLine; ShowText(s);

procedure Stroke;
   Stroke the path

procedure TextOut(X, Y: Single; const Text: PDFString);
   Show some text at a specified page position

procedure TextOutW(X, Y: Single; PW: PWideChar);
   Show some unicode text at a specified page position

procedure TextRect(ARect: TPdfRect; const Text: PDFString; Alignment:
   TPdfAlignment; Clipping: boolean);
   Show the text in the specified rectangle and alignment
   - optional clipping can be applied

property Contents: TPdfStream read FContents;
   Retrieve the current Canvas content stream, i.e. where the PDF commands are to be written to

property Doc: TPdfDocument read GetDoc;
   Retrieve the associated PDF document instance which created this Canvas

property Page: TPdfPage read GetPage;
   Retrieve the current Canvas Page
property RightToLeftText: Boolean read fRightToLeftText write fRightToLeftText;
   If Uniscribe-related methods must handle the text from right to left

TPdfDictionaryWrapper = class(TPersistent)
   Common ancestor to all dictionary wrapper classes
   property Data: TPdfDictionary read FData write SetData;
      The associated dictionary, containing all data
   property HasData: boolean read GetHasData;
      Return TRUE if has any data stored within

TPdfInfo = class(TPdfDictionaryWrapper)
   A dictionary wrapper class for the PDF document information fields
   - all values use the generic VCL string type, and will be encoded as Unicode if necessary
   property Author: string read GetAuthor write SetAuthor;
      The PDF document Author
   property CreationDate: TDateTime read GetCreationDate write SetCreationDate;
      The PDF document Creation Date
   property Creator: string read GetCreator write SetCreator;
      The Software or Library name which created this PDF document
   property Keywords: string read GetKeywords write SetKeywords;
      The PDF document associated key words
   property ModDate: TDateTime read GetModDate write SetModDate;
      The PDF document modification date
   property Subject: string read GetSubject write SetSubject;
      The PDF document subject
   property Title: string read GetTitle write SetTitle;
      The PDF document title

TPdfCatalog = class(TPdfDictionaryWrapper)
   A dictionary wrapper class for the PDF document catalog fields
   - It contains references to other objects defining the document’s contents, outline, article threads (PDF 1.1), named destinations, and other attributes. In addition, it contains information about how the document should be displayed on the screen, such as whether its outline and thumbnail page images should be displayed automatically and whether some location other than the first page should be shown when the document is opened
   property NonFullScreenPageMode: TPdfPageMode read GetNonFullScreenPageMode write SetNonFullScreenPageMode;
      Page mode determines how the document should appear when opened
property OpenAction: TPdfDestination read FOpenAction write FOpenAction;
A Destination to be displayed when the document is opened

property PageLayout: TPdfPageLayout read GetPageLayout write SetPageLayout;
The page layout to be used when the document is opened

property PageMode: TPdfPageMode read GetPageMode write SetPageMode;
Page mode determines how the document should appear when opened

property Pages: TPdfDictionary read GetPages write SetPages;
The page tree node that is the root of the document's page tree
- Required, must be an indirect reference
- you can set a value to it in order to add some nested pages

property ViewerPreference: TPdfViewerPreferences read GetViewerPreference write SetViewerPreference;
A viewer preferences dictionary specifying the way the document is to be displayed on the screen
- If this entry is absent, viewer applications should use their own current user preference settings

TPdfFont = class(TPdfDictionaryWrapper)
A generic PDF font object

constructor Create(AXref: TPdfXref; const AName: PDFString);
Create the PDF font object instance

function GetAnsiCharWidth(const AText: PDFString; APos: integer): integer; virtual;
Retrieve the width of a specified character
- implementation of this method is either WinAnsi (by TPdfFontWinAnsi), either compatible with MBCS strings (TPdfFontCIDFontType2)
- return 0 by default (descendant must handle the Ansi charset)

procedure AddUsedWinAnsiChar(aChar: AnsiChar);
Mark some WinAnsi char as used

property Name: PDFString read FName;
The internal PDF font name (e.g. 'Helvetica-Bold')
- postscript font names are inside the unit: these postscript names could not match the "official" True Type font name, stored as UTF-8 in FTrueTypeFonts

property ShortCut: PDFString read FShortCut;
The internal PDF shortcut (e.g. 'F3')

property Unicode: boolean read fUnicode;
Is set to TRUE if the font is dedicated to Unicode Chars

TPdfFontWinAnsi = class(TPdfFont)
A generic PDF font object, handling at least WinAnsi encoding
- TPdfFontTrueType descendant will handle also Unicode chars, for all WideChar which are outside the WinAnsi selection
**Synopse mORMot Framework**

**Software Architecture Design 1.18**  
**Date: September 16, 2020**

---

**SynPdf.pas unit**  
Rev. 1.18  
Page 1505 of 2552

```pascal

destructor Destroy; override;

Release the used memory

function GetAnsiCharWidth(const AText: PDFString; APos: integer): integer; override;

Retrieve the width of a specified character
- implementation of this method expect WinAnsi encoding
- return the value contained in fWinAnsiWidth[] by default

TPdfFontType1 = class(TPdfFontWinAnsi)

An embedded WinAnsi-Encoded standard Type 1 font
- handle Helvetica, Courier and Times font by now

constructor Create(AXref: TPdfXref; const AName: PDFString; WidthArray: PSmallIntArray); reintroduce;

Create a standard font instance, with a given name and char widths
- if WidthArray is nil, it will create a fixed-width font of 600 units

TPdfFontCIDFontType2 = class(TPdfFont)

An embedded Composite CIDFontType2
- i.e. a CIDFont whose glyph descriptions are based on TrueType font technology
- typically handle Japan or Chinese standard fonts
- used with MBCS encoding, not WinAnsi

TPdfTTF = class(TObject)

Handle Unicode glyph description for a True Type Font
- cf http://www.microsoft.com/typography/OTSPEC/otff.htm#otttables
- handle Microsoft cmap format 4 encoding (i.e. most used true type fonts on Windows)

endCode: PWordArray;

End characterCode for each cmap format 4 segment

fmt4: ^TCmapFmt4;

Character to glyph mapping (cmap) table, in format 4

glyphIndexArray: PWordArray;

Glyph index array (arbitrary length)

head: ^TCmapHEAD;

These are pointers to the useful data of the True Type Font: Font header

hhea: ^TCmapHHEA;

Horizontal header

idDelta: PSmallIntArray;

Delta for all character codes in each cmap format 4 segment

idRangeOffset: PWordArray;

Offsets into glyphIndexArray or 0
```
startCode: PWordArray;

    Start character code for each cmap format 4 segment

constructor Create(aUnicodeTTF: TPdfFontTrueType); reintroduce;

    Create Unicode glyph description for a supplied True Type Font
    - the HDC of its corresponding document must have selected the font first
    - this constructor will fill fUsedWide[] and fUsedWideChar of aUnicodeTTF with every available
      unicode value, and its corresponding glyph and width

TPdfFontTrueType = class(TPdfFontWinAnsi)

    Handle TrueType Font
    - handle both WinAnsi text and Unicode characters in two separate TPdfFontTrueType instances
      (since PDF need two separate fonts with diverse encoding)

constructor Create(ADoc: TPdfDocument; AFontIndex: integer; AStyle: TPdfFontStyles; const ALogFont: TLogFontW; AWinAnsiFont: TPdfFontTrueType); reintroduce;

    Create the TrueType font object instance

destructor Destroy; override;

    Release the associated memory and handles

function FindOrAddUsedWideChar(aWideChar: WideChar): integer;

    Mark some UTF-16 codepoint as used
    - return the index in fUsedWideChar[] and fUsedWide[]
    - this index is the one just added, or the existing one if the value was found to be already in the
      fUserWideChar[] array

function GetWideCharWidth(aWideChar: WideChar): Integer;

    Retrieve the width of an UTF-16 codepoint
    - WinAnsi characters are taken from fWinAnsiWidth[], unicode chars from fUsedWide[].Width

property FixedWidth: boolean read fFixedWidth;

    Is set to TRUE if the font has a fixed width

property Style: TPdfFontStyles read fStyle;

    The associated Font Styles

property UnicodeFont: TPdfFontTrueType read fUnicodeFont;

    Points to the corresponding Unicode font
    - returns NIL if the Unicode font has not yet been created by the CreateUnicodeFont method
    - may return SELF if the font is itself the Unicode version

property WideCharUsed: Boolean read GetWideCharUsed;

    Is set to TRUE if the PDF used any true type encoding

property WinAnsiFont: TPdfFontTrueType read fWinAnsiFont;

    Points to the corresponding WinAnsi font
    - always return a value, whatever it is self
TPdfDestination = class(TObject)

A destination defines a particular view of a document, consisting of the following:
- The page of the document to be displayed
- The location of the display window on that page
- The zoom factor to use when displaying the page

constructor Create(APdfDoc: TPdfDocument);

Create the PDF destination object
- the current document page is associated with this destination

destructor Destroy; override;

Release the object

function GetValue: TPdfArray;

Retrieve the array containing the location of the display window
- the properties values which are not used are ignored

property Bottom: Integer index 3 read GetElement write SetElement;

Retrieve the bottom coordinate of the location of the display window

property DestinationType: TPdfDestinationType read FType write FType;

Destination Type determines default user space coordinate system of Explicit destinations

property Doc: TPdfDocument read FDoc;

The associated PDF document which created this Destination object

property Left: Integer index 0 read GetElement write SetElement;

Retrieve the left coordinate of the location of the display window

property Page: TPdfPage read FPage;

The associated Page

property PageHeight: Integer read GetPageHeight;

The page height of the current page
- return the corresponding MediaBox value

property PageWidth: Integer read GetPageWidth;

The page width of the current page
- return the corresponding MediaBox value

property Reference: TObject read FReference write FReference;

An object associated to this destination, to be used for convenience

property Right: Integer index 2 read GetElement write SetElement;

Retrieve the right coordinate of the location of the display window

property Top: Integer index 1 read GetElement write SetElement;

Retrieve the top coordinate of the location of the display window

property Zoom: Single read FZoom write SetZoom;

The associated Zoom factor
- by default, the Zoom factor is 1
TPdfOutlineEntry = class(TPdfDictionaryWrapper)

An Outline entry in the PDF document

constructor Create(AParent: TPdfOutlineEntry; TopPosition: integer=-1);
reintroduce;

Create the Outline entry instance
- if TopPosition is set, a corresponding destination is created on the current PDF Canvas page, at this Y position

destructor Destroy; override;

Release the associated memory and reference object

function AddChild(TopPosition: integer=-1): TPdfOutlineEntry;

Create a new entry in the outline tree
- this is the main method to create a new entry

property Dest: TPdfDestination read FDest write FDest;

The associated destination

property Doc: TPdfDocument read FDoc;

The associated PDF document which created this Destination object

property First: TPdfOutlineEntry read FFirst;

The first outline entry of this entry list

property Last: TPdfOutlineEntry read FLast;

The last outline entry of this entry list

property Level: integer read FLevel write FLevel;

An internal property (not exported to PDF content)

property Next: TPdfOutlineEntry read FNext;

The next outline entry of this entry

property Opened: boolean read FOpened write FOpened;

If the outline must be opened

property Parent: TPdfOutlineEntry read FParent;

The parent outline entry of this entry

property Prev: TPdfOutlineEntry read FPrev;

The previous outline entry of this entry

property Reference: TObject read FReference write FReference;

An object associated to this destination, to be used for convenience

property Title: string read FTitle write FTitle;

The associated title
- is a generic VCL string, so is Unicode ready
TPdfOutlineRoot = class(TPdfOutlineEntry)

Root entry for all Outlines of the PDF document
- this is a "fake" entry which must be used as parent for all true TPdfOutlineEntry instances, but must not be used as a true outline entry

constructor Create(ADoc: TPdfDocument); reintroduce;

Create the Root entry for all Outlines of the PDF document

procedure Save; override;

Update internal parameters (like outline entries count) before saving

TPdfPageGDI = class(TPdfPage)

A PDF page, with its corresponding Meta File and Canvas

destructor Destroy; override;

Release associated memory

TPdfDocumentGDI = class(TPdfDocument)

Class handling PDF document creation using GDI commands
- this class allows using a VCL standard Canvas class
- handles also PDF creation directly from TMetaFile content

constructor Create(AUseOutlines: Boolean=false; ACodePage: integer=0; APDFA1: boolean=false ; AEncryption: TPdfEncryption=nil);

Create the PDF document instance, with a VCL Canvas property
- see TPdfDocument.Create constructor for the arguments expectations

function AddPage: TPdfPage; override;

Add a Page to the current PDF document

procedure SaveToStream(AStream: TStream; ForceModDate: TDateTime=0); override;

Save the PDF file content into a specified Stream
- this overridden method draw first the all VCLCanvas content into the PDF
procedure SaveToStreamDirectPageFlush(FlushCurrentPageNow: boolean=false); override;

Save the current page content to the PDF file
- this overridden method flush the content from the VCLCanvas into the PDF
- it will reduce the used memory as much as possible, by-passing page content compression
- typical use may be:

```pascal
with TPdfDocumentGDI.Create do
try
  Stream := TFileStream.Create(FileName, fmCreate);
  try
    SaveToStreamDirectBegin(Stream);
    for i := 1 to 9 do
      begin
        AddPage;
        with VCLCanvas do
          begin
            Font.Name := 'Times new roman';
            Font.Size := 150;
            Font.Style := [fsBold, fsItalic];
            Font.Color := clNavy;
            TextOut(100, 100, 'Page ' + IntToStr(i));
          end;
        SaveToStreamDirectPageFlush; // direct writing
      end;
    SaveToStreamDirectEnd;
  finally
    Stream.Free;
  end;
finally
  Free;
end;
```

property KerningHScaleBottom: Single read fKerningHScaleBottom write fKerningHScaleBottom;

The % limit below which Font Kerning is transformed into PDF Horizontal Scaling commands (when text positioning is tpKerningFromAveragePosition)
- set to 99.0 by default

property KerningHScaleTop: Single read fKerningHScaleTop write fKerningHScaleTop;

The % limit over which Font Kerning is transformed into PDF Horizontal Scaling commands (when text positioning is tpKerningFromAveragePosition)
- set to 101.0 by default

property UseMetaFileTextClipping: TPdfCanvasRenderMetaFileTextClipping read fUseMetaFileTextClipping write fUseMetaFileTextClipping;

Defines how TMetaFile text clipping should be applied
- tcNeverClip has been reported to work better e.g. when app is running on Wine

property UseMetaFileTextPositioning: TPdfCanvasRenderMetaFileTextPositioning read fUseMetaFileTextPositioning write fUseMetaFileTextPositioning;

Defines how TMetaFile text positioning is rendered
- default is tpSetTextJustification
- tpSetTextJustification if content used SetTextJustification() API calls
- tpExactTextCharacterPositining for exact font kerning, but resulting in bigger pdf size
- tpKerningFromAveragePosition will compute average pdf Horizontal Scaling in association with KerningHScaleBottom/KerningHScaleTop properties
- replace deprecated property UseSetTextJustification
property VCLCanvas: TCanvas read GetVCLCanvas;
  The VCL Canvas of the current page

property VCLCanvasSize: TSize read GetVCLCanvasSize;
  The VCL Canvas size of the current page
  - useful to calculate coordinates for the current page
  - filled with (0,0) before first call to VCLCanvas property

TPdfImage = class(TPdfXObject)
  Generic image object
  - is either bitmap encoded or jpeg encoded

constructor Create(aDoc: TPdfDocument; aImage: TGraphic; DontAddToFXref: boolean);
  Create the image from a supplied VCL TGraphic instance
  - handle TBitmap and SynGdiPlus picture types, i.e. TJpegImage (stored as jpeg), and
    TGifImage/TPngImage (stored as bitmap)
  - use TPdfForm to handle TMetafile in vectorial format
  - an optional DontAddToFXref is available, if you don't want to add this object to the main XRef
    list of the PDF file

constructor CreateJpegDirect(aDoc: TPdfDocument; aImageFile: TMemoryStream;
  DontAddToFXref: boolean=true); reintroduce; overload;
  Create an image from a supplied JPEG content
  - an optional DontAddToFXref is available, if you don't want to add this object to the main XRef
    list of the PDF file

constructor CreateJpegDirect(aDoc: TPdfDocument; const aJpegFileName: TFileName;
  DontAddToFXref: boolean=true); reintroduce; overload;
  Create an image from a supplied JPEG file name
  - will raise an EFOpenError exception if the file doesn't exist
  - an optional DontAddToFXref is available, if you don't want to add this object to the main XRef
    list of the PDF file

property PixelHeight: Integer read fPixelHeight;
  Height of the image, in pixels units

property PixelWidth: Integer read fPixelWidth;
  Width of the image, in pixels units

TPdfForm = class(TPdfXObject)
  Handle any form XObject
  - A form XObject (see Section 4.9, of PDF reference 1.3) is a self-contained description of an
    arbitrary sequence of graphics objects, defined as a PDF content stream

constructor Create(aDoc: TPdfDocumentGDI; aMetaFile: TMetafile); reintroduce;
  Create a form XObject from a supplied TMetaFile
TPdfFormWithCanvas = class(TPdfXObject)

A form XObject with a Canvas for drawing
- once created, you can create this XObject, then draw it anywhere on any page - see sample

constructor Create(aDoc: TPdfDocument; W, H: Integer); reintroduce;
Create a form XObject with TPDFCanvas

destructor Destroy; override;
Release used memory

procedure CloseCanvas;
Close the internal canvas

property Canvas: TPdfCanvas read FCanvas;
Access to the private canvas associated with the PDF form XObject

TPdfObjectStream = class(TPdfXObject)

Used to handle compressed object stream (in PDF 1.5 format)

constructor Create(aDoc: TPdfDocument); reintroduce;
Create the instance, i.e. its associated stream

destructor Destroy; override;
Release internal memory structures

function AddObject(Value: TPdfObject): integer;
Add an object to this compressed object stream
- returns the object index in this object stream

property ObjectCount: integer read fObjectCount;
The number of compressed objects within this object stream

TScriptState = packed record

An UniScribe script state
- uBidiLevel: Unicode Bidi algorithm embedding level (0..16)
- fFlags: Script state flags

TScriptAnalysis = packed record

An Uniscribe script analysis
- eScript: Shaping engine
- fFlags: Script analysis flags
- s: Script state

TScriptItem = packed record

A Uniscribe script item, after analysis of a unicode text

a: TScriptAnalysis;
Corresponding Uniscribe script analysis
iCharPos: Integer;
   Logical offset to first character in this item

TScriptProperties = packed record
   Contains information about Uniscribe special processing for each script
   fFlags: TScriptProperties_set;
      Set of possible Uniscribe processing properties for a given language
   langId: Word;
      Primary and sublanguage associated with script

TScriptVisAttr = packed record
   Contains the visual (glyph) attributes that identify clusters and justification points, as generated by ScriptShape
   uJustification: Justification class
   fFlags: Uniscribe visual (glyph) attributes
   fShapeReserved: Reserved for use by shaping engines

Types implemented in the SynPdf unit

PDFString = AnsiString;
   The PDF library use internally AnsiString text encoding
   - the corresponding charset is the current system charset, or the one supplied as a parameter to TPdfDocument.Create

TCmapSubTableArray = packed array[byte] of packed record
   platformID: word;
   platformSpecificID: word;
   offset: Cardinal;
end;
   Points to every ‘cmap’ encoding subtables

TGradientDirection = ( gdHorizontal, gdVertical );
   PDF gradient direction

TLineCapStyle = ( lcButt_End, lcRound_End, lcProjectingSquareEnd );
   Line cap style specifies the shape to be used at the ends of open subpaths when they are stroked

TLineJoinStyle = ( ljMiterJoin, ljRoundJoin, ljBevelJoin );
   The line join style specifies the shape to be used at the corners of paths that are stroked

TPdfAlignment = ( paLeftJustify, paRightJustify, paCenter );
   PDF text paragraph alignment

TPdfAnnotationBorder = ( abSolid, abDashed, abBeveled, abInset, abUnderline );
   The border style of an annotation

TPdfAnnotationSubType = ( asTextNotes, asLink );
   The annotation types determines the valid annotation subtype of TPdfDoc

TPdfBuffer32 = array[0..31] of byte;
   Internal 32 bytes buffer, used during encryption process

TPdfCanvasArcType = ( acArc, acArcTo, acArcAngle, acPie, acChoord );
   Is used to define the TMetaFile kind of arc to be drawn
TPdfCanvasRenderMetaFileTextClipping = ( tcClipReference, tcClipExplicit, tcAlwaysClip, tcNeverClip );

*Is used to define how TMetaFile text is clipped*
- by default, text will be clipped with the specified TEMRText.ptlReference
- you could set tcClipExplicit to clip following the specified rclBounds
- or tcAlwaysClip to use the current clipping region (if any)
- finally, tcNeverClip would disable whole text clipping process, which has been reported to be preferred e.g. on Wine

TPdfCanvasRenderMetaFileTextPositioning = ( tpKerningFromAveragePosition, tpSetTextJustification, tpExactTextCharacterPositioning );

*Is used to define how TMetaFile text positioning is rendered*
- tpSetTextJustification will handle efficiently the fact that TMetaFileCanvas used SetTextJustification() API calls to justify text: it will converted to SetWordSpace() pdf rendering
- tpExactTextCharacterPositioning will use the individual glyph positioning information as specified within the TMetaFile content: resulting pdf size will be bigger, but font kerning will be rendered as expected
- tpKerningFromAveragePosition will use global font kerning via SetHorizontalScaling() pdf rendering

TPdfColor = -$7FFFFFFF-1..$7FFFFFFF;

*The available PDF color range*

TPdfColorRGB = cardinal;

*The PDF color, as expressed in RGB terms*
- maps COLORREF / TColorRef as used e.g. under windows

TPdfCompressionMethod = ( cmNone, cmFlateDecode );

*Define if streams must be compressed*

TPdfDate = PDFString;

*A PDF date, encoded as 'D:20100414113241'*

TPdfDestinationType = ( dtXYZ, dtFit, dtFitH, dtFitV, dtFitR, dtFitB, dtFitBH, dtFitBV );

*Destination Type determines default user space coordinate system of Explicit destinations*

TPdfEncryptionLevel = ( elNone, elRC4_40, elRC4_128 );

*The available encryption levels*
- in current version only RC4 40-bit and RC4 128-bit are available, which correspond respectively to PDF 1.3 and PDF 1.4 formats
- for RC4 40-bit and RC4 128-bit, associated password are restricted to a maximum length of 32 characters and could contain only characters from the Latin-1 encoding (i.e. no accent)

TPdfEncryptionPermission = ( epPrinting, epGeneralEditing, epContentCopy, epAuthoringComment, epFillingForms, epContentExtraction, epDocumentAssembly, epPrintingHighResolution );

*PDF can encode various restrictions on document operations which can be granted or denied individually (some settings depend on others, though):*
- Printing: If printing is not allowed, the print button in Acrobat will be disabled. Acrobat supports a distinction between high-resolution and low-resolution printing. Low-resolution printing generates a bitmapped image of the page which is suitable only for personal use, but prevents high-quality reproduction and re-distilling. Note that bitmap printing not only results in low output quality, but will also considerably slow down the printing process.
- General Editing: If this is disabled, any document modification is prohibited. Content extraction and
printing are allowed.
- Content Copying and Extraction: If this is disabled, selecting document contents and copying it to
the clipboard for repurposing the contents is prohibited. The accessibility interface also is disabled. If
you need to search such documents with Acrobat you must select the Certified Plugins Only
preference in Acrobat.
- Authoring Comments and Form Fields: If this is disabled, adding, modifying, or deleting comments
and form fields is prohibited. Form field filling is allowed.
- Form Field Fill-in or Signing: If this is enabled, users can sign and fill in forms, but not create form
fields.
- Document Assembly: If this is disabled, inserting, deleting or rotating pages, or creating bookmarks
and thumbnails is prohibited.

```pascal
TPdfEncryptionPermissions = set of TPdfEncryptionPermission;
Set of restrictions on PDF document operations
```

```pascal
TPdfFileFormat = ( pdf13, pdf14, pdf15, pdf16 );
The internal pdf file format
```

```pascal
TPdfFontStandard = ( pfsTimes, pfsHelvetica, pfsCourier );
The recognized families of the Standard 14 Fonts
```

```pascal
TPdfFontStyle = ( pfsBold, pfsItalic, pfsUnderline, pfsStrikeOut );
Potential font styles
```

```pascal
TPdfFontStyles = set of TPdfFontStyle;
Set of font styles
```

```pascal
TPdfGDIComment = ( pgcOutline, pgcBookmark, pgcLink, pgcLinkNoBorder );
Defines the data stored inside a EMR_GDICOMMENT message
- pgcOutline can be used to add an outline at the current position (i.e. the last Y parameter of a
  Move): the text is the associated title, UTF-8 encoded and the outline tree is created from the
  number of leading spaces in the title
- pgcBookmark will create a destination at the current position (i.e. the last Y parameter of a Move),
  with some text supplied as bookmark name
- pgcLink/pgcLinkNoBorder will create a asLink annotation, expecting the data to be filled with TRect
  inclusive-inclusive bounding rectangle coordinates, followed by the corresponding bookmark name
  - use the GDIComment*() functions to append the corresponding EMR_GDICOMMENT message to a
    metafile content
```

```pascal
TPdfObjectType = ( otDirectObject, otIndirectObject, otVirtualObject );
Allowed types for PDF objects (i.e. TPdfObject)
```

```pascal
TPdfPageLayout = ( plSinglePage, plOneColumn, plTwoColumnLeft, plTwoColumnRight );
The page layout to be used when the document is opened
```

```pascal
TPdfPageMode = ( pmUseNone, pmUseOutlines, pmUseThumbs, pmFullScreen );
Page mode determines how the document should appear when opened
```

```pascal
TPDFPageSize = ( psA4, psA5, psA3, psA2, psA1, psA0, psLetter, psLegal, psUserDefined );
Available known paper size (psA4 is the default on TPdfDocument creation)
```

```pascal
TPdfViewerPreference = ( vpHideToolbar, vpHideMenuBar, vpHideWindowUI, vpFitWindow, vpCenterWindow, vpEnforcePrintScaling );
```
Viewer preferences specifying how the reader User Interface must start
- vpEnforcePrintScaling will set the file version to be PDF 1.6

TPdfViewerPreferences = set of TPdfViewerPreference;
Set of Viewer preferences

TPScriptPropertiesArray = array[byte] of PScriptProperties;
An array of Uniscribe processing information

TScriptAnalysis_enum = ( s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, fRTL, fLayoutRTL, fLinkBefore, fLinkAfter, fLogicalOrder, fNoGlyphIndex );
Uniscribe script analysis flag elements
- s0,s1,s2,s3,s4,s5,s6,s7,s8,s9: map TScriptAnalysis.eScript
- fRTL: Rendering direction
- fLayoutRTL: Set for GCP classes ARABIC/HEBREW and LOCALNUMBER
- fLinkBefore: Implies there was a ZWJ before this item
- fLinkAfter: Implies there is a ZWJ following this item.
- fLogicalOrder: Set by client as input to ScriptShape/Place
- fNoGlyphIndex: Generated by ScriptShape/Place - this item does not use glyph indices

TScriptAnalysis_set = set of TScriptAnalysis_enum;
A set of Uniscribe script analysis flags

TScriptProperties_enum = ( fNumeric, fComplex, fNeedsWordBreaking, fNeedsCareInfo, bCharSet0, bCharSet1, bCharSet2, bCharSet3, bCharSet4, bCharSet5, bCharSet6, bCharSet7, fControl, fPrivateUseArea, fNeedsCharacterJustify, fInvalidGlyph, fInvalidLogAttr, fCDM, fAmbiguousCharSet, fClusterSizeVaries, fRejectInvalid );
All possible Uniscribe processing properties of a given language
- fNumeric: if a script contains only digits
- fComplex: Script requires special shaping or layout
- fNeedsWordBreaking: Requires ScriptBreak for word breaking information
- fNeedsCareInfo: Requires caret restriction to cluster boundaries
- bCharSet0 .. bCharSet7: Charset to use when creating font
- fControl: Contains only control characters
- fPrivateUseArea: This item is from the Unicode range U+E000 through U+F8FF
- fNeedsCharacterJustify: Requires inter-character justification
- fInvalidGlyph: Invalid combinations generate glyph wgInvalid in the glyph buffer
- fInvalidLogAttr: Invalid combinations are marked by fInvalid in the logical attributes
- fCDM: Contains Combining Diacritical Marks
- fAmbiguousCharSet: Script does not correspond 1/1:1 with a charset
- fClusterSizeVaries: Measured cluster width depends on adjacent clusters
- fRejectInvalid: Invalid combinations should be rejected

TScriptProperties_set = set of TScriptProperties_enum;
Set of possible Uniscribe processing properties of a given language

TScriptState_enum = ( r0, r1, r2, r3, r4, fOverrideDirection, fInhibitSymSwap, fCharShape, fDigitSubstitute, fInhibitLigate, fDisplayZWG, fArabicNumContext, fGcpClusters );
UniScribe script state flag elements
- r0,r1,r2,r3,r4: map TScriptState.uBidiLevel
- fOverrideDirection: Set when in LRO/RLO embedding
- fInhibitSymSwap: Set by U+206A (ISS), cleared by U+206B (ASS)
- fCharShape: Set by U+206D (AAFS), cleared by U+206C (IAFS)
- fDigitSubstitute: Set by U+206E (NADS), cleared by U+206F (NODS)
- fInhibitLigate: Equiv !GCP_Ligate, no Unicode control chars yet
- fDisplayZWG: Equiv GCP_DisplayZWG, no Unicode control characters yet
- fArabicNumContext: For EN->AN Unicode rule
- fGcpClusters: For Generating Backward Compatible GCP Clusters (legacy Apps)

TScriptState_set = set of TScriptState_enum;
A set of UniScribe script state flags

TScriptVisAttr_enum = ( a0, a1, a2, a3, fClusterStart, fDiacritic, fZeroWidth, fReserved );
Uniscribe visual (glyph) attributes
- a0 .. a3: map the Justification class number
- fClusterStart: First glyph of representation of cluster
- fDiacritic: Diacritic
- fZeroWidth: Blank, ZWJ, ZWNJ etc, with no width
- fReserved: General reserved bit

TScriptVisAttr_set = set of TScriptVisAttr_enum;
Set of Uniscribe visual (glyph) attributes

TTextRenderingMode = ( trFill, trStroke, trFillThenStroke, trInvisible, trFillClipping,
trStrokeClipping, trFillStrokeClipping, trClipping );
The text rendering mode determines whether text is stroked, filled, or used as a clipping path

TUsedWide = array of packed record case byte of 0: ( Width: word; Glyph: word; ); 1:
( Int: integer; ); end;
This dynamic array stores details about used unicode characters
- every used unicode character has its own width and glyph index in the true type font content

TXObjectId = integer;
Numerical ID for every XObject

Constants implemented in the SynPdf unit

CRLF = #10;
The Carriage Return and Line Feed values used in the PDF file generation
- expect #13 and #10 under Windows, but #10 (e.g. only Line Feed) is enough for the PDF standard,
and will create somewhat smaller PDF files

LF = #10;
The Line Feed value

MSWINDOWS_DEFAULT_FONTS: RawUTF8 = 'Arial'#13#10'Courier New'#13#10'Georgia'#13#10+'Impact'#13#10'Lucida Console'#13#10'Roman'#13#10'Symbol'#13#10+'Tahoma'#13#10'Times New Roman'#13#10'Trebuchet'#13#10+'Verdana'#13#10'WingDings';
List of common fonts available by default since Windows 2000
- to not embed these fonts in the PDF document, and save some KB, just use the
EmbeddedTTFIgnore property of TPdfDocument/TPdfDocumentGDI:
- note that this is useful only if the EmbeddedTTF property was set to TRUE

PDF_FREE_ENTRY = 'f';
Used for an unused (free) xref entry, e.g. the root entry

\[
\text{PDF\_IN\_USE\_ENTRY} = 'n';
\]

Used for an used xref entry

\[
\text{PDF\_MAX\_GENERATION\_NUM} = 65535;
\]

Used e.g. for the root xref entry

\[
\text{PDF\_PERMISSION\_ALL: TPdfEncryptionPermissions} = \\
\{\text{Low(TPdfEncryptionPermission)}..\text{high(TPdfEncryptionPermission)}\};
\]

- Allow all actions for a pdf encrypted file
- to be used as parameter for TPdfEncryption.New() class method

\[
\text{PDF\_PERMISSION\_NOCOPY: TPdfEncryptionPermissions} = \{\text{epPrinting}, \text{epAuthoringComment}, \text{epPrintingHighResolution}, \text{epFillingForms}\};
\]

- Disable content extraction or copy for a pdf encrypted file
- to be used as parameter for TPdfEncryption.New() class method

\[
\text{PDF\_PERMISSION\_NOCOPYNORPRINT: TPdfEncryptionPermissions} = [];
\]

- Disable printing and content extraction or copy for a pdf encrypted file
- to be used as parameter for TPdfEncryption.New() class method

\[
\text{PDF\_PERMISSION\_NOMODIF: TPdfEncryptionPermissions} = \{\text{epPrinting}, \text{epContentCopy}, \text{epPrintingHighResolution}, \text{epFillingForms}, \text{epContentExtraction}, \text{epDocumentAssembly}\};
\]

- Disable modification and annotation of a pdf encrypted file
- to be used as parameter for TPdfEncryption.New() class method

\[
\text{PDF\_PERMISSION\_NOPRINT: TPdfEncryptionPermissions} = \{\text{epGeneralEditing}, \text{epContentCopy}, \text{epAuthoringComment}, \text{epContentExtraction}, \text{epDocumentAssembly}\};
\]

- Disable printing for a pdf encrypted file
- to be used as parameter for TPdfEncryption.New() class method

USP\_E\_SCRIPT\_NOT\_IN\_FONT = HRESULT((SEVERITY\_ERROR shl 31) or (FACILITY\_ITF shl 16)) or $200;

Error returned by Uniscribe when the current selected font does not contain sufficient glyphs or shaping tables

Functions or procedures implemented in the SynPdf unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrentPrinterPaperSize</td>
<td>Retrieve the paper size used by the current selected printer</td>
<td>1518</td>
</tr>
<tr>
<td>CurrentPrinterResolution</td>
<td>Retrieve the current printer resolution</td>
<td>1518</td>
</tr>
<tr>
<td>GDICommentBookmark</td>
<td>Append a EMR_GDICOMMENT message for handling PDF bookmarks</td>
<td>1518</td>
</tr>
<tr>
<td>GDICommentLink</td>
<td>Append a EMR_GDICOMMENT message for creating a Link into a specified bookmark</td>
<td>1518</td>
</tr>
<tr>
<td>GDICommentOutline</td>
<td>Append a EMR_GDICOMMENT message for handling PDF outline</td>
<td>1518</td>
</tr>
<tr>
<td>L2R</td>
<td>Reverse char orders for every hebrew and arabic words</td>
<td>1519</td>
</tr>
<tr>
<td>PdfBox</td>
<td>Wrapper to create a temporary PDF box</td>
<td>1519</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>PdfCoord</td>
<td>Convert some millimeters dimension to internal PDF twips value</td>
<td>1519</td>
</tr>
<tr>
<td>PdfRect</td>
<td>Wrapper to create a temporary PDF coordinates rectangle</td>
<td>1519</td>
</tr>
<tr>
<td>PdfRect</td>
<td>Wrapper to create a temporary PDF coordinates rectangle</td>
<td>1519</td>
</tr>
<tr>
<td>RawUTF8ToPDFString</td>
<td>Convert a specified UTF-8 content into a PDFString value</td>
<td>1519</td>
</tr>
<tr>
<td>ScriptApplyDigitSubstitution</td>
<td>Uniscribe function to apply the specified digit substitution settings to the specified script control and script state structures</td>
<td>1519</td>
</tr>
<tr>
<td>ScriptGetProperties</td>
<td>Uniscribe function to retrieve information about the current scripts</td>
<td>1519</td>
</tr>
<tr>
<td>ScriptItemize</td>
<td>Uniscribe function to break a Unicode string into individually shapeable items</td>
<td>1519</td>
</tr>
<tr>
<td>ScriptLayout</td>
<td>Uniscribe function to convert an array of run embedding levels to a map of visual-to-logical position and/or logical-to-visual position</td>
<td>1520</td>
</tr>
<tr>
<td>ScriptShape</td>
<td>Uniscribe function to generate glyphs and visual attributes for an Unicode run</td>
<td>1520</td>
</tr>
<tr>
<td>UInt32ToPDFString</td>
<td>Convert an unsigned integer into a PDFString text</td>
<td>1520</td>
</tr>
<tr>
<td>_DateTimeToPdfDate</td>
<td>Convert a date, into PDF string format, i.e. as 'D:20100414113241Z'</td>
<td>1520</td>
</tr>
<tr>
<td>_HasMultiByteString</td>
<td>This function returns TRUE if the supplied text contain any MBCS character</td>
<td>1520</td>
</tr>
<tr>
<td>_PdfDateToDateTine</td>
<td>Decode PDF date, encoded as 'D:20100414113241'</td>
<td>1520</td>
</tr>
</tbody>
</table>

**function** CurrentPrinterPaperSize: TPDFPaperSize;
*Retrieve the paper size used by the current selected printer*

**function** CurrentPrinterRes: TPoint;
*Retrieve the current printer resolution*

**procedure** GDICommentBookmark(MetaHandle: HDC; const aBookmarkName: RawUTF8);
*Append a EMR_GDICOMMENT message for handling PDF bookmarks*
- will create a PDF destination at the current position (i.e. the last Y parameter of a Move), with some text supplied as bookmark name

**procedure** GDICommentLink(MetaHandle: HDC; const aBookmarkName: RawUTF8; const aRect: TRect; NoBorder: boolean);
*Append a EMR_GDICOMMENT message for creating a Link into a specified bookmark*

**procedure** GDICommentOutline(MetaHandle: HDC; const aTitle: RawUTF8; aLevel: Integer);
*Append a EMR_GDICOMMENT message for handling PDF outline*
- used to add an outline at the current position (i.e. the last Y parameter of a Move): the text is the associated title, UTF-8 encoded and the outline tree is created from the specified numerical level (0=root)
procedure L2R(W: PWideChar; L: integer);
    Reverse char orders for every hebrew and arabic words
    - just reverse all the UTF-16 codepoints in the supplied buffer

definition PdfBox(Left, Top, Width, Height: Single): TPdfBox;
    Wrapper to create a temporary PDF box

definition PdfCoord(MM: single): integer;
    Convert some milli meters dimension to internal PDF twips value

definition PdfRect(Left, Top, Right, Bottom: Single): TPdfRect; overload;
    Wrapper to create a temporary PDF coordinates rectangle

definition PdfRect(const Box: TPdfBox): TPdfRect; overload;
    Wrapper to create a temporary PDF coordinates rectangle

definition RawUTF8ToPDFString(const Value: RawUTF8): PDFString;
    Convert a specified UTF-8 content into a PDFString value

definition ScriptApplyDigitSubstitution(const psds: Pointer; const psControl: pointer; const psState: pointer): HRESULT;
    Uniscribe function to apply the specified digit substitution settings to the specified script control and script state structures

    Uniscribe function to retrieve information about the current scripts
    - ppSp: Pointer to an array of pointers to SCRIPT_PROPERTIES structures indexed by script.
    - piNumScripts: Pointer to the number of scripts. The valid range for this value is 0 through piNumScripts-1.

definition ScriptItemize(const pwcInChars: PWideChar; cInChars: Integer; cMaxItems: Integer; const psControl: pointer; const psState: pointer; pItems: PScriptItem; var pcItems: Integer): HRESULT;
    Uniscribe function to break a Unicode string into individually shapeable items
    - pwcInChars: Pointer to a Unicode string to itemize.
    - cInChars: Number of characters in pwcInChars to itemize.
    - cMaxItems: Maximum number of SCRIPT_ITEM structures defining items to process.
    - psControl: Optional. Pointer to a SCRIPT_ITEM structure indicating the type of itemization to perform. Alternatively, the application can set this parameter to NULL if no SCRIPT_ITEM properties are needed.
    - psState: Optional. Pointer to a SCRIPT_STATE structure indicating the initial bidirectional algorithm state. Alternatively, the application can set this parameter to NULL if the script state is not needed.
    - pItems: Pointer to a buffer in which the function retrieves SCRIPT_ITEM structures representing the items that have been processed. The buffer should be cMaxItems*sizeof(SCRIPT_ITEM) + 1 bytes in length. It is invalid to call this function with a buffer to hold less than two SCRIPT_ITEM structures. The function always adds a terminal item to the item analysis array so that the length of the item with zero-based index "i" is always available as:
        pItems[i+1].iCharPos - pItems[i].iCharPos
    - pcItems: Pointer to the number of SCRIPT_ITEM structures processed
function ScriptLayout(cRuns: Integer; const pbLevel: PByte; piVisualToLogical: PInteger; piLogicalToVisual: PInteger): HRESULT; stdcall; external Usp10;

Uniscribe function to convert an array of run embedding levels to a map of visual-to-logical position and/or logical-to-visual position
- cRuns: Number of runs to process
- pbLevel: Array of run embedding levels
- piVisualToLogical: List of run indices in visual order
- piLogicalToVisual: List of visual run positions

function ScriptShape(hdc: HDC; var psc: pointer; const pwcChars: PWideChar; cChars: Integer; cMaxGlyphs: Integer; psa: PScriptAnalysis; pwOutGlyphs: PWord; pwLogClust: PWord; psva: PScriptVisAttr; var pcGlyphs: Integer): HRESULT; stdcall; external Usp10;

Uniscribe function to generate glyphs and visual attributes for an Unicode run
- hdc: Optional (see under caching)
- psc: Uniscribe font metric cache handle
- pwcChars: Logical unicode run
- cChars: Length of unicode run
- cMaxGlyphs: Max glyphs to generate
- psa: Result of ScriptItemize (may have fNoGlyphIndex set)
- pwOutGlyphs: Output glyph buffer
- pwLogClust: Logical clusters
- psva: Visual glyph attributes
- pcGlyphs: Count of glyphs generated

function UInt32ToPDFString(Value: Cardinal): PDFString;

Convert an unsigned integer into a PDFString text

function _DateTimeToPdfDate(ADate: TDateTime): TPdfDate;

Convert a date, into PDF string format, i.e. as 'D:20100414113241Z'

function _HasMultiByteString(Value: PAnsiChar): boolean;

This function returns TRUE if the supplied text contain any MBCS character
- typical call must check first if MBCS is currently enabled
  if SysLocale.FarEast and _HasMultiByteString(pointer(Text)) then ...

function _PdfDateToDateTime(const AText: TPdfDate): TDateTime;

Decode PDF date, encoded as 'D:20100414113241'
27.30. SynProtoRelay.pas unit

*Purpose*: Implements asynchronous safe WebSockets tunnelling
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynProtoRelay unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynBidirSock</td>
<td>Implements bidirectional client and server protocol, e.g. WebSockets - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>680</td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
<tr>
<td>SynCrtSock</td>
<td>Classes implementing TCP/UDP/HTTP client and server protocol - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1083</td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher) - implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT - optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support) - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1139</td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1363</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features - as a complement to SynCommons, which tended to increase too much - licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1721</td>
</tr>
<tr>
<td>SynWinSock</td>
<td>Low level access to network Sockets for the Win32 platform - this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1843</td>
</tr>
</tbody>
</table>
## SynProtoRelay class hierarchy

![SynProtoRelay class hierarchy diagram](image)

### Objects implemented in the SynProtoRelay unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPrivateRelay</td>
<td>Implements a Private Relay client, located in a private network behind a restricted firewall</td>
<td>1524</td>
</tr>
<tr>
<td>TPublicRelay</td>
<td>Implements a Public Relay server, e.g. located on a small Linux/BSD box</td>
<td>1524</td>
</tr>
<tr>
<td>TRelayClientProtocol</td>
<td>Private Relay to Public Relay connection, decapsulating connection ID</td>
<td>1523</td>
</tr>
<tr>
<td>TRelayFrame</td>
<td>Internal binary serialized content for frames tunnelling</td>
<td>1523</td>
</tr>
<tr>
<td>TRelayFrameRestPayload</td>
<td>TRelayFrame.payload for opcode = focRestPayload = focReservedF</td>
<td>1522</td>
</tr>
<tr>
<td>TRelayServerProtocol</td>
<td>Public Relay to Private Relay connection, encapsulating connection ID</td>
<td>1523</td>
</tr>
<tr>
<td>TServerClient</td>
<td>Implements a relayed link to the local ORM/SOA server instance</td>
<td>1524</td>
</tr>
<tr>
<td>TSynopseClientProtocol</td>
<td>Private Relay to local Server connection</td>
<td>1523</td>
</tr>
<tr>
<td>TSynopseServerProtocol</td>
<td>Regular mORMot client to Public Relay connection using synopsejson/synopsebin/synopsebinary protocols</td>
<td>1523</td>
</tr>
</tbody>
</table>

**TRelayFrameRestPayload = packed record**

TRelayFrame.payload for opcode = focRestPayload = focReservedF

**TRelayFrame = packed record**

Internal binary serialized content for frames tunnelling
Synopse mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

**TSynopseServerProtocol = class (TWebSocketProtocol)**

*Regular mORMot client to Public Relay connection using synopsejson/synopsebin/synopsebinary protocols*
- any incoming frame will be encapsulated with the connection ID, then relayed to the Private Relay node using TRelayServerProtocol

**constructor Create(aOwner: TPublicRelay); reintroduce;**
*Initialize the protocol to be processed on a given TPublicRelay*

**function Clone(const aClientURI: RawUTF8): TWebSocketProtocol; override;**
*Used server-side for any new connection*

**TRelayServerProtocol = class (TWebSocketProtocol)**

*Public Relay to Private Relay connection, encapsulating connection ID*
- see also TRelayClientProtocol as the reversed process
- any incoming frame will be decapsulate the associated connection ID, then relayed to the proper client node using TSynopseServerProtocol

**constructor Create(aOwner: TPublicRelay; const aServerKey: RawUTF8); reintroduce;**
*Initialize the protocol to be processed on a given TPublicRelay*

**function Clone(const aClientURI: RawUTF8): TWebSocketProtocol; override;**
*Used server-side for any new connection*

**TRelayClientProtocol = class (TWebSocketProtocol)**

*Private Relay to Public Relay connection, decapsulating connection ID*
- see also TRelayServerProtocol as the reversed process

**constructor Create(aOwner: TPrivateRelay; const aRelayKey: RawUTF8); reintroduce;**
*Initialize the protocol to be processed on a given TPrivateRelay*

**function Clone(const aClientURI: RawUTF8): TWebSocketProtocol; override;**
*Used server-side for any new connection*

**TSynopseClientProtocol = class (TWebSocketProtocol)**

*Private Relay to local Server connection*
- forward raw frames from TSynopseServerProtocol

**constructor Create(aOwner: TPrivateRelay; const aProtocolName: RawUTF8); reintroduce;**
*Initialize the protocol to be processed on a given TPrivateRelay*
- the protocol is relayed from TRelayClientProtocol.ProcessIncomingFrame

**function Clone(const aClientURI: RawUTF8): TWebSocketProtocol; override;**
*Used server-side for any new connection*
TPublicRelay = class(TAbstractRelay)

*Implements a Public Relay server, e.g. located on a small Linux/BSD box*

**constructor** Create(aLog: TSynLogClass; const aClientsPort, aServerPort: SockString;
const aServerKey: RawUTF8; aServerJWT: TJWTAbstract; aClientsThreadPoolCount: integer=2; aClientsKeepAliveTimeOut: integer=30000); *reintroduce*

*Initialize the Public Relay*

- WebSockets clients will connect to aClientsPort as usual
- all communication will be encapsulated and relayed to aServerPort, using the optional
  aServerKey for TWebSocketProtocol.SetEncryptKey(), and aServerJWT to authenticate the
  incoming connection (owned by this instance)

**destructor** Destroy; *override*

*Finalize the Public Relay server*

**property** Clients: TWebSocketServer read fClients;

*Raw access to the ORM/SOA clients server instance*

**property** Server: TWebSocketServer read fServer;

*Raw access to the Private Relay server instance*

**property** ServerJWT: TJWTAbstract read fServerJWT;

*Access to the JWT authentication for TPrivateRelay communication*

TServerClient = class(THttpClientWebSockets)

*Implements a relayed link to the local ORM/SOA server instance*

- add some internal fields for TPrivateRelay.fServers[]

TPrivateRelay = class(TAbstractRelay)

*Implements a Private Relay client, located in a private network behind a restricted firewall*

**constructor** Create(aLog: TSynLogClass; const aRelayHost, aRelayPort, aRelayKey,
aRelayBearer, aServerHost, aServerPort, aServerRemoteIPHeader: RawUTF8);

*reintroduce*

*Initialize the Private Relay*

- communication will be encapsulated and relayed to aServerHost/aServerPort via a new
  TServerClient connection, using aServerWebSocketsURI, aServerWebSocketsEncryptionKey,
aServerWebSocketsAJAX and aServerWebSocketsCompression parameters as any regular client
  in the local network - from the server point of view, those clients will appear like local clients
  unless ServerRemoteIPHeader is set according to the
  TSQLHttpRequestDefinition.ServerRemoteIPHeader value (e.g. as 'X-Real-IP') and the remote
  client IP will be used instead
- Connected/TryConnect should be called on a regular basis to connect to the Public Relay using
  aRelayHost/aRelayPort/aRelayKey/aRelayBearer

**destructor** Destroy; *override*

*Finalize the Private Relay server*

**function** Connected: boolean;

*Check if the Public Relay did connect to this Private Relay instance*
function Encrypted: boolean;
   True if this Private Relay uses encryption with the Public Relay

function TryConnect: boolean;
   (re)connect to aRelayHost/aRelayPort Public Relay via a single link
   - will first disconnect is needed

procedure Disconnect;
   Disconnect from aRelayHost/aRelayPort Public Relay

property Connections: integer read fServersCount;
   How many client connections are actually relayed via this instance

property RelayConnected: boolean read Connected;
   True if the Private Relay is connected to the Public Relay

property RelayEncrypted: boolean read Encrypted;
   True if the Private Relay to the Public Relay link is encrypted

property RelayHost: RawUTF8 read fRelayHost;
   Public Relay host address

property RelayPort: RawUTF8 read fRelayPort;
   Public Relay host port

property ServerHost: RawUTF8 read fServerHost;
   Local processing server host address

property ServerPort: RawUTF8 read fServerPort;
   Local processing server host port

Constants implemented in the SynProtoRelay unit

foRestPayload = focReservedF;
   If TRelayFrame.payload is in fact a TRelayFrameRestPayload

RELAYFRAME_VER = $aa00;
   As set to TRelayFrame
27.31. SynProtoRTSPHTTP.pas unit

Purpose: Implements asynchronous RTSP stream tunnelling over HTTP
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the SynProtoRTSPHTTP unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynBidirSock</td>
<td>Implements bidirectional client and server protocol, e.g. WebSockets</td>
<td>680</td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynCrtSock</td>
<td>Classes implementing TCP/UDP/HTTP client and server protocol</td>
<td>1083</td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td>SynTests</td>
<td>Unit test functions used by Synopse projects</td>
<td>1832</td>
</tr>
<tr>
<td>SynWinSock</td>
<td>Low level access to network Sockets for the Win32 platform</td>
<td>1843</td>
</tr>
</tbody>
</table>

SynProtoRTSPHTTP class hierarchy

Objects implemented in the SynProtoRTSPHTTP unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPostConnection</td>
<td>Holds a HTTP POST connection for RTSP proxy</td>
<td>1527</td>
</tr>
<tr>
<td>TRtspConnection</td>
<td>Holds a RTSP connection for HTTP GET proxy</td>
<td>1527</td>
</tr>
<tr>
<td>TRTSPOverHTTPServer</td>
<td>Implements RTSP over HTTP asynchronous proxy</td>
<td>1527</td>
</tr>
</tbody>
</table>
TPostConnection = class(TAsynchConnection)
  Holds a HTTP POST connection for RTSP proxy
  - as used by the TRTSPOverHTTPServer class

TRtspConnection = class(TAsynchConnection)
  Holds a RTSP connection for HTTP GET proxy
  - as used by the TRTSPOverHTTPServer class

TRTSPOverHTTPServer = class(TAsynchServer)
  Implements RTSP over HTTP asynchronous proxy
  - the HTTP transport is built from two separate HTTP GET and POST requests initiated by the client; the server then binds the connections to form a virtual full-duplex connection - see https://goo.gl/CX6VA3 for reference material about this horrible, but widely accepted, Apple hack

constructor Create(const aRtspServer, aRtspPort, aHttpPort: SockString; aLog: TSynLogClass; aOnStart, aOnStop: TNotifyThreadEvent; aOptions: TAsynchConnectionsOptions = []); reintroduce;
  Initialize the proxy HTTP server forwarding specified RTSP server:port

destructor Destroy; override;
  Shutdown and finalize the server

function HttpToRtsp(const HttpURI: RawUTF8): RawUTF8;
  Convert a http://... proxy URI into a rtsp://.... URI

function RtspToHttp(const RtspURI: RawUTF8): RawUTF8;
  Convert a rtsp://.... URI into a http://... proxy URI
  - will reuse the rtsp public server name, but change protocol to http:// and set the port to RtspPort

procedure RegressionTests(test: TSynTestCase; clientcount, steps: integer);
  Perform some basic regression and benchmark testing on a running server

property HttpPort: SockString read GetHttpPort;
  The bound HTTP port

property RtspPort: SockString read fRtspPort;
  The associated RTSP server port

property RtspServer: SockString read fRtspServer;
  The associated RTSP server address
27.32. SynSelfTests.pas unit

Purpose: Automated tests for common units of the Synopse mORMot Framework
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The SynSelfTests unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.2.2</td>
<td>The framework libraries, including all its SQLite3 related features, shall be tested using Unitary testing</td>
<td>2549</td>
</tr>
</tbody>
</table>

Units used in the SynSelfTests unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dddDomAuthInterfaces</td>
<td>Shared DDD Domains: Authentication objects and interfaces - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2437</td>
</tr>
<tr>
<td>dddDomCountry</td>
<td>Shared DDD Domains: TCountry object definition - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2439</td>
</tr>
<tr>
<td>dddDomUserinterfaces</td>
<td>Shared DDD Domains: User interfaces definition - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2445</td>
</tr>
<tr>
<td>dddDomUserTypes</td>
<td>Shared DDD Domains: User objects definition - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2447</td>
</tr>
<tr>
<td>dddInfraAuthRest</td>
<td>Shared DDD Infrastructure: Authentication implementation - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2463</td>
</tr>
<tr>
<td>dddInfraEmail</td>
<td>Shared DDD Infrastructure: implement an email validation service - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2467</td>
</tr>
<tr>
<td>dddInfraEmailer</td>
<td>Shared DDD Infrastructure: generic emailing service - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2471</td>
</tr>
<tr>
<td>dddInfraRepoUser</td>
<td>Shared DDD Infrastructure: User CQRS Repository via ORM - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2475</td>
</tr>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1899</td>
</tr>
<tr>
<td>mORMotDB</td>
<td>Virtual Tables for external DB access for mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2276</td>
</tr>
<tr>
<td>Unit Name</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
</tbody>
</table>
| mORMotDDD             | Domain-Driven-Design toolbox for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                       | 2285  |
| mORMotHttpClient      | HTTP/1.1 RESTful JSON Client classes for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                               | 2306  |
| mORMotHttpServer      | HTTP/1.1 RESTFUL JSON Server classes for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                 | 2313  |
| mORMotMongoDB         | Direct optimized MongoDB access for mORMot's ORM  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                     | 2334  |
| mORMotMVC             | Implements MVC patterns over mORMot's ORM/SOA and SynMustache  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                 | 2339  |
| mORMotService         | Daemon management classes for mORMot, including low-level Win NT Service  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                 | 2370  |
| mORMotSQLite3         | SQLite3 embedded Database engine used as the mORMot SQL kernel  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                     | 2382  |
| PasZip                 | ZIP/LZ77 Deflate/Inflate Compression in pure pascal  
- this unit is a part of the freeware Synopse framework, licensed in the LGPL v3; version 1.18                                                                                                          | 674   |
| SynBidirSock           | Implements bidirectional client and server protocol, e.g. WebSockets  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                 | 680   |
| SynCommons             | Common functions used by most Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                       | 717   |
| SynCrtSock             | Classes implementing TCP/UDP/HTTP client and server protocol  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                     | 1083  |
| SynCrypto              | Fast cryptographic routines (hashing and cypher)  
- implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT  
- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                 | 1139  |
<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynDBODBC</td>
<td>ODBC 3.x library direct access classes to be used with our SynDB architecture</td>
<td>1274</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynDBRemote</td>
<td>Remote access to any RDBMS via HTTP using our SynDB architecture</td>
<td>1293</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynDBSQLite3</td>
<td>SQLite3 direct access classes to be used with our SynDB architecture</td>
<td>1298</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynEcc</td>
<td>Certificate-based public-key cryptography using ECC-secp256r1</td>
<td>1313</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynGdiPlus</td>
<td>GDI+ library API access</td>
<td>1350</td>
</tr>
<tr>
<td></td>
<td>- adds GIF, TIF, PNG and JPG pictures read/write support as standard</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TGraphic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- make available most useful GDI+ drawing methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- allows Antialiased rending of any EMF file using GDI+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLizard</td>
<td>Lizard (LZ5) compression routines (statically linked for FPC)</td>
<td>1359</td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; original Lizard is BSD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2-Clause</td>
<td></td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLZ</td>
<td>SynLZ Compression routines</td>
<td>1393</td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLZO</td>
<td>Fast LZO Compression routines</td>
<td>1395</td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.13</td>
<td></td>
</tr>
<tr>
<td>SynMongoDB</td>
<td>MongoDB document-oriented database direct access classes</td>
<td>1396</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynMustache</td>
<td>Logic-less mustache template rendering</td>
<td>1450</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynOleDB</td>
<td>Fast OleDB direct access classes</td>
<td>1458</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>
## Synopse mORMot Framework

Software Architecture Design 1.18

Date: September 16, 2020

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynPdf</td>
<td>PDF file generation&lt;br&gt;- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1473</td>
</tr>
<tr>
<td>SynProtoRelay</td>
<td>Implements asynchronous safe WebSockets tunnelling&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1521</td>
</tr>
<tr>
<td>SynProtoRTSPHTTP</td>
<td>Implements asynchronous RTSP stream tunnelling over HTTP&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1526</td>
</tr>
<tr>
<td>SynSQLite3</td>
<td>SQLite3 Database engine direct access&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1646</td>
</tr>
<tr>
<td>SynSQLite3Static</td>
<td>SQLite3 3.33.0 Database engine - statically linked for Windows/Linux&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1711</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features&lt;br&gt;- as a complement to SynCommons, which tended to increase too much&lt;br&gt;- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1721</td>
</tr>
<tr>
<td>SynTests</td>
<td>Unit test functions used by Synopse projects&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1832</td>
</tr>
<tr>
<td>SynZip</td>
<td>Low-level access to ZLib compression (1.2.5 engine version)&lt;br&gt;- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1845</td>
</tr>
<tr>
<td>TestSQL3FPCInterfaces</td>
<td>SOA interface methods definition to circumvent FPC missing RTTI&lt;br&gt;- generated at 2016-06-14 13:49:41</td>
<td>2436</td>
</tr>
</tbody>
</table>
**SynSelfTests class hierarchy**

## Objects implemented in the **SynSelfTests** unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBidirCallback</td>
<td>SOA callback definition as expected by TTestBidirectionalRemoteConnection</td>
<td>1547</td>
</tr>
<tr>
<td>IBidirService</td>
<td>SOA service definition as expected by TTestBidirectionalRemoteConnection</td>
<td>1547</td>
</tr>
<tr>
<td>ICalculator</td>
<td>A test interface, used by TTestServiceOrientedArchitecture</td>
<td>1549</td>
</tr>
<tr>
<td>IComplexCalculator</td>
<td>A test interface, used by TTestServiceOrientedArchitecture</td>
<td>1550</td>
</tr>
<tr>
<td>IComplexNumber</td>
<td>A test interface, used by TTestServiceOrientedArchitecture</td>
<td>1551</td>
</tr>
<tr>
<td>IDDDThreadsCommand</td>
<td>CQRS Command Interface for TTest</td>
<td>1548</td>
</tr>
<tr>
<td>IDDDThreadsQuery</td>
<td>CQRS Query Interface for TTest</td>
<td>1548</td>
</tr>
<tr>
<td>ISmsSender</td>
<td>A test interface, used by TTestServiceOrientedArchitecture</td>
<td>1551</td>
</tr>
<tr>
<td>ITestGroup</td>
<td>A test interface, used by TTestServiceOrientedArchitecture</td>
<td>1551</td>
</tr>
<tr>
<td>ITestPerThread</td>
<td>A test interface, used by TTestServiceOrientedArchitecture</td>
<td>1551</td>
</tr>
<tr>
<td>ITestSession</td>
<td>A test interface, used by TTestServiceOrientedArchitecture</td>
<td>1551</td>
</tr>
<tr>
<td>ITestUser</td>
<td>A test interface, used by TTestServiceOrientedArchitecture</td>
<td>1551</td>
</tr>
<tr>
<td>IUserRepository</td>
<td>A test interface, used by TTestServiceOrientedArchitecture</td>
<td>1551</td>
</tr>
<tr>
<td>TComplexNumber</td>
<td>A test class, used by TTestServiceOrientedArchitecture</td>
<td>1549</td>
</tr>
<tr>
<td>TConsultaNav</td>
<td>A record used by IComplexCalculator.EchoRecord</td>
<td>1549</td>
</tr>
<tr>
<td>TCustomerData</td>
<td>A record used by IComplexCalculator.GetCustomer</td>
<td>1549</td>
</tr>
<tr>
<td>TDDDTest</td>
<td>This is our simple Test data class. Will be mapped to TSQLRecordDDDTest.</td>
<td>1548</td>
</tr>
<tr>
<td>TSQLRecordDDDTest</td>
<td>The corresponding TSQLRecord for TDDDTest.</td>
<td>1548</td>
</tr>
<tr>
<td>TSQLRecordPeople</td>
<td>Under Linux, port&lt;1024 needs root user a record mapping used in the test classes of the framework</td>
<td>1535</td>
</tr>
<tr>
<td>TSQLRestServerTest</td>
<td>This class defined two published methods of type TSQLRestServerCallBack in order to test the Server-Side ModelRoot/TableName/ID/MethodName RESTful model</td>
<td>1544</td>
</tr>
<tr>
<td>TTestBasicClasses</td>
<td>This test case will test some generic classes defined and implemented in the mORMot.pas unit</td>
<td>1539</td>
</tr>
<tr>
<td>TTestBidirectionalRemoteConnection</td>
<td>A test case for all bidirectional remote access, e.g. WebSockets</td>
<td>1547</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>TTestClientServerAccess</td>
<td>This test case will test most functions, classes and types defined and implemented in the mORMotSQLite3 unit, i.e. the SQLite3 engine itself, used as a HTTP/1.1 server and client</td>
<td>1543</td>
</tr>
<tr>
<td>TTestCompression</td>
<td>This test case will test most functions, classes and types defined and implemented in the SynZip unit</td>
<td>1539</td>
</tr>
<tr>
<td>TTestCryptographicRoutines</td>
<td>This test case will test most functions, classes and types defined and implemented in the SynCrypto unit</td>
<td>1539</td>
</tr>
<tr>
<td>TTestDDDMultiThread</td>
<td>A test case for aggressive multi-threaded DDD ORM test</td>
<td>1549</td>
</tr>
<tr>
<td>TTestDDDSharedUnits</td>
<td>A test case for all shared DDD types and services</td>
<td>1548</td>
</tr>
<tr>
<td>TTestECCCryptography</td>
<td>This test case will test ECDH and ECDSA cryptography as implemented in the SynECC unit</td>
<td>1540</td>
</tr>
<tr>
<td>TTestExternalDatabase</td>
<td>A test case which will test most external DB functions of the mORMotDB unit</td>
<td>1545</td>
</tr>
<tr>
<td>TTestFileBased</td>
<td>This test case will test most functions, classes and types defined and implemented in the mORMotSQLite3 unit, i.e. the SQLite3 engine itself, with a file-based approach</td>
<td>1542</td>
</tr>
<tr>
<td>TTestFileBasedMemoryMap</td>
<td>This test case will test most functions, classes and types defined and implemented in the mORMotSQLite3 unit, i.e. the SQLite3 engine itself, with a file-based approach</td>
<td>1543</td>
</tr>
<tr>
<td>TTestFileBasedWAL</td>
<td>This test case will test most functions, classes and types defined and implemented in the mORMotSQLite3 unit, i.e. the SQLite3 engine itself, with a file-based approach</td>
<td>1543</td>
</tr>
<tr>
<td>TTestLowLevelCommon</td>
<td>This test case will test most functions, classes and types defined and implemented in the SynCommons unit</td>
<td>1535</td>
</tr>
<tr>
<td>TTestLowLevelTypes</td>
<td>This test case will test most low-level functions, classes and types defined and implemented in the mORMot.pas unit</td>
<td>1538</td>
</tr>
<tr>
<td>TTestMemoryBased</td>
<td>This test case will test most functions, classes and types defined and implemented in the mORMotSQLite3 unit, i.e. the SQLite3 engine itself, with a memory-based approach</td>
<td>1542</td>
</tr>
<tr>
<td>TTestMultiThreadProcess</td>
<td>A test case for multi-threading abilities of the framework</td>
<td>1546</td>
</tr>
<tr>
<td>TTestProtocols</td>
<td>This test case will validate several low-level protocols</td>
<td>1541</td>
</tr>
<tr>
<td>TTestServiceOrientedArchitecture</td>
<td>A test case which will test the interface-based SOA implementation of the mORMot framework</td>
<td>1552</td>
</tr>
<tr>
<td>TTestSQLite3Engine</td>
<td>A parent test case which will test most functions, classes and types defined and implemented in the mORMotSQLite3 unit, i.e. the SQLite3 engine itself</td>
<td>1541</td>
</tr>
<tr>
<td>TTestSynopsePDF</td>
<td>This test case will test most functions, classes and types defined and implemented in the SynPDF unit</td>
<td>1541</td>
</tr>
</tbody>
</table>
**Objects** | **Description** | **Page**
---|---|---
TUser | A test value object, used by IUserRepository/ISmsSender interfaces | 1551

---

**TSQLRecordPeople** = class(TSQLRecord)

*Under Linux, port<1024 needs root user a record mapping used in the test classes of the framework*
- this class can be used for debugging purposes, with the database created by TTestFileBased in mORMotSQLite3.pas
- this class will use 'People' as a table name

**function** DataAsHex(aClient: TSQLRestClientURI): RawUTF8;

*Method used to test the Client Side ModelRoot/TableName/ID/MethodName RESTful request, i.e. ModelRoot/People/ID/DataAsHex in this case*
- this method calls the supplied TSQLRestClient to retrieve its results, with the ID taken from the current TSQLRecordPeople instance ID field
- parameters and result types depends on the purpose of the function
- TSQLRestServerTest.DataAsHex published method implements the result calculation on the Server-Side

**class function** Sum(aClient: TSQLRestClientURI; a, b: double; Method2: boolean): double;

*Method used to test the Client Side ModelRoot/MethodName RESTful request, i.e. ModelRoot/Sum in this case*
- this method calls the supplied TSQLRestClient to retrieve its results
- parameters and result types depends on the purpose of the function
- TSQLRestServerTest.Sum published method implements the result calculation on the Server-Side
- this method doesn't expect any ID to be supplied, therefore will be called as class function - normally, it should be implement in a TSQLRestClient descendant, and not as a TSQLRecord, since it doesn't depend on TSQLRecordPeople at all
- you could also call the same service from the ModelRoot/People/ID/Sum URL, but it won't make any difference

---

**TTestLowLevelCommon** = class(TSynTestCase)

*This test case will test most functions, classes and types defined and implemented in the SynCommons unit*

*Used for DI-2.2.2 (page 2549).*

**procedure** BaudotCode;

*Test ASCII Baudot encoding*

**procedure** Bits;

*The low-level bit management functions*

**procedure** BloomFilters;

*Test TSynBloomFilter class*
procedure Curr64;
  The new fast Currency to/from string conversion

procedure CustomRTL;
  Validate our optimized MoveFast/FillCharFast functions

procedure FastStringCompare;
  Test StrComp() and AnsiComp() functions

procedure IniFiles;
  The fast .ini file content direct access

procedure Integers;
  Test low-level integer/Int64 functions

procedure Iso8601DateAndTime;
  The ISO-8601 date and time encoding
  - test especially the conversion to/from text

procedure MimeTypes;
  Test mime types recognition

procedure NumericalConversions;
  Low level fast Integer or Floating-Point to/from string conversion
  - especially the RawUTF8 or PUTF8Char relative versions

procedure QuickSelect;
  Validates the median computation using the "Quick Select" algorithm

procedure Soundex;
  The Soundex search feature (i.e. TSynSoundex and all related functions)

procedure SystemCopyRecord;
  The faster CopyRecord function, enhancing the system.pas unit

procedure TimeZones;
  Test the TSynTimeZone class and its cross-platform local time process

procedure UrlDecoding;
  Test UrlEncode() and UrlDecode() functions
  - this method use some ISO-8601 encoded dates and times for the testing

procedure UrlEncoding;
  Test UrlEncode() and UrlDecode() functions

procedure _CamelCase;
  The camel-case / camel-uncase features, used for i18n from Delphi RTII

procedure _crc32c;
  Test crc32c in both software and hardware (SSE4.2) implementations

procedure _DeltaCompress;
  Test DeltaCompress/DeltaExtract functions

procedure _GUID;
  Test our internal fast TGUID process functions
procedure _IdemPropName;
  Test IdemPropName() and IdemPropNameU() functions

procedure _IsMatch;
  Test IsMatch() function

procedure _ParseCommandArguments;
  Test ParseCommandArguments() function

procedure _Random32;
  Test RDRAND Intel x86/x64 opcode if available, or fast gsl_rng_taus2

procedure _TDynArray;
  Test the TDynArray object and methods

procedure _TDynArrayHashed;
  Test the TDynArrayHashed object and methods (dictionary features)
  - this test will create an array of 200,000 items to test speed

procedure _TExprParserMatch;
  Test TExprParserMatch class

procedure _TObjArray;
  Test T*ObjArray types and the ObjArray*() wrappers

procedure _TObjectDynArrayWrapper;
  Test TObjectDynArrayWrapper class

procedure _TObjectListHashed;
  Test TObjectListHashed class

procedure _TObjectListSorted;
  Test TObjectListSorted class

procedure _TRawUTF8Interning;
  Test TRawUTF8Interning process

procedure _TRawUTF8List;
  Test the TRawUTF8List class

procedure _TSynCache;
  Test the TSynCache class

procedure _TSynDictionary;
  Test the TSynDictionary class

procedure _TSynFilter;
  Low-level TSynFilter classes

procedure _TSynLogFile;
  Low-level TSynLogFile class

procedure _TSynNameValue;
  Test TSynNameValue class
procedure _TSynQueue;
    Validate the TSynQueue class

procedure _TSynTable;
    Test TSynTable class and TSynTableVariantType new variant type

procedure _TSynUniqueIdentifier;
    Client side genuine 64 bit identifiers generation

procedure _TSynValidate;
    Low-level TSynValidate classes

procedure _UTF8;
    Test UTF-8 and Win-Ansi conversion (from or to, through RawUnicode)

TTestLowLevelTypes = class(TSyntestCase)
    This test case will test most low-level functions, classes and types defined and implemented in the mORMot.pas unit

    Used for DI-2.2.2 (page 2549).

procedure EncodeDecodeJSON;
    Some low-level JSON encoding/decoding

procedure MustacheRenderer;
    Test the Mustache template rendering unit

procedure RTTI;
    Some low-level RTTI access
    - especially the field type retrieval from published properties

procedure UrlEncoding;
    Some low-level Url encoding from parameters

procedure Variants;
    Some low-level variant process

procedure WikiMarkdownToHtml;
    HTML generation from Wiki Or Markdown syntax

procedure _BSON;
    BSON process (using TDocVariant)

procedure _TDecimal128;
    Low-level TDecimal128 decimal value process (as used in BSON)

procedure _TDocVariant;
    Variant-based JSON/BSON document process

procedure _TSynMonitorUsage;
    Test advanced statistics monitoring

procedure _TSynTableStatement;
    Test SELECT statement parsing
**TTestBasicClasses** = **class**(TSynTestCase)

This test case will test some generic classes defined and implemented in the mORMot.pas unit

*Used for DI-2.2.2 (page 2549).*

**procedure** _TSQLModel;

Test the TSQLModel class

**procedure** _TSQLRecord;

Test the TSQLRecord class
- especially SQL auto generation, or JSON export/import

**procedure** _TSQLRecordSigned;

Test the digital signature of records

**procedure** _TSQLRestServerFullMemory;

Test a full in-memory server over Windows Messages
- Under Linux, URIDll will be used instead due to lack of message loop
- without any SQLite3 engine linked

**TTestCompression** = **class**(TSynTestCase)

This test case will test most functions, classes and types defined and implemented in the SynZip unit

**procedure** GZIPFormat;

.gzip archive handling

**procedure** InMemoryCompression;

Direct deflate/inflate functions

**procedure** ZIPFormat;

.zip archive handling

**procedure** _SynLZ;

SynLZ internal format

**procedure** _SynLZO;

SynLZO internal format

**procedure** _TAlgoCompress;

TAlgoCompress classes

**TTestCryptographicRoutines** = **class**(TSynTestCase)

This test case will test most functions, classes and types defined and implemented in the SynCrypto unit

**procedure** Benchmark;

Compute some performance numbers, mostly against regression

**procedure** _Adler32;

Adler32 hashing functions
procedure _AES256;
AES encryption/decryption functions

procedure _AES_GCM;
AES-GCM encryption/decryption with authentication

procedure _Base64;
Base-64 encoding/decoding functions

procedure _CompressShaAes;
CompressShaAes() using SHA-256 / AES-256-CTR algorithm over SynLZ

procedure _CryptDataForCurrentUser;
CryptDataForCurrentUser() function

procedure _CryptDataForCurrentUserAPI;
CryptDataForCurrentUserAPI() function

procedure _JWT;
JWT classes

procedure _MD5;
MD5 hashing functions

procedure _RC4;
RC4 encryption function

procedure _SHA1;
SHA-1 hashing functions

procedure _SHA256;
SHA-256 hashing functions

procedure _SHA3;
SHA-3 / Keccak hashing functions

procedure _SHA512;
SHA-512 hashing functions

procedure _TAESPNRG;
AES-based pseudorandom number generator

TTestECCCryptography = class(TSynTestCase)
This test case will test ECDH and ECDSA cryptography as implemented in the SynECC unit

procedure CertificatesAndSignatures;
ECDSA certificates chains and digital signatures

procedure ECCCommandLineTool;
Run most commands of the ECC tool

procedure ECDHEStreamProtocol;
ECDHE stream protocol
<table>
<thead>
<tr>
<th>Procedure Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReferenceVectors;</td>
<td>Avoid regression among platforms and compilers</td>
</tr>
<tr>
<td>_ecc_make_key;</td>
<td>ECC private/public keys generation</td>
</tr>
<tr>
<td>_ecdh_shared_secret;</td>
<td>ECDH key derivation</td>
</tr>
<tr>
<td>_ecdsa_sign;</td>
<td>ECDSA signature computation</td>
</tr>
<tr>
<td>_ecdsa_verify;</td>
<td>ECDSA signature verification</td>
</tr>
</tbody>
</table>

**TTestProtocols = class(TSynTestCase)**

This test case will validate several low-level protocols

<table>
<thead>
<tr>
<th>Procedure Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTSPOverHTTP;</td>
<td>RTSP over HTTP, as implemented in SynProtoRTSPHTTP unit</td>
</tr>
</tbody>
</table>

**TTestSynopsePDF = class(TSynTestCase)**

This test case will test most functions, classes and types defined and implemented in the SynPDF unit

<table>
<thead>
<tr>
<th>Procedure Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_TPdfDocument;</td>
<td>Create a PDF document, using the PDF Canvas property</td>
</tr>
<tr>
<td></td>
<td>- test font handling, especially standard font substitution</td>
</tr>
<tr>
<td>_TPdfDocumentGDI;</td>
<td>Create a PDF document, using a EMF content</td>
</tr>
<tr>
<td></td>
<td>- validates the EMF/TMetaFile enumeration, and its conversion into the PDF content, including PDF-1.5 and page orientation</td>
</tr>
<tr>
<td></td>
<td>- this method will produce a .pdf file in the executable directory, if you want to check out the result (it's simply a curve drawing, with data from NIST)</td>
</tr>
</tbody>
</table>

**TTestSQLite3Engine = class(TSynTestCase)**

A parent test case which will test most functions, classes and types defined and implemented in the mORMotSQLite3 unit, i.e. the SQLite3 engine itself

- it should not be called directly, but through TTestFileBased, TTestMemoryBased and TTestMemoryBased children

*Used for DI-2.2.2 (page 2549).*

<table>
<thead>
<tr>
<th>Procedure Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatabaseDirectAccess;</td>
<td>Test direct access to the SQLite3 engine</td>
</tr>
<tr>
<td></td>
<td>- i.e. via TSQLDataBase and TSQLRequest classes</td>
</tr>
<tr>
<td>VirtualTableDirectAccess;</td>
<td>Test direct access to the Virtual Table features of SQLite3</td>
</tr>
</tbody>
</table>
procedure _TRecordVersion;
Test Master/Slave replication using TRecordVersion field

procedure _TSQLRestClientDB;
Test the TSQLRestClientDB, i.e. a local Client/Server driven usage of the framework
- validates TSQLModel, TSQLRestServer and TSQLRestStorage by checking the coherency of the data between client and server instances, after update from both sides
- use all RESTful commands (GET/UPDATE/POST/DELETE...)
- test the 'many to many' features (i.e. TSQLRecordMany) and dynamic arrays published properties handling
- test dynamic tables

procedure _TSQLTableJSON;
Test the TSQLTableJSON table
- the JSON content generated must match the original data
- a VACCUM is performed, for testing some low-level SQLite3 engine implementation
- the SortField feature is also tested

TTestFileBased = class(TTestSQLite3Engine)
This test case will test most functions, classes and types defined and implemented in the mORMotSQLite3 unit, i.e. the SQLite3 engine itself, with a file-based approach
Used for DI-2.2.2 (page 2549).

TTestMemoryBased = class(TTestSQLite3Engine)
This test case will test most functions, classes and types defined and implemented in the mORMotSQLite3 unit, i.e. the SQLite3 engine itself, with a memory-based approach
- this class will also test the TSQLRestStorage class, and its 100% Delphi simple database engine
Used for DI-2.2.2 (page 2549).

procedure ShardRead;
Validate TSQLRestStorageShardDB reading among all sharded databases

procedure ShardReadAfterPurge;
Validate TSQLRestStorageShardDB reading after deletion of several shards

procedure ShardWrite;
Validate TSQLRestStorageShardDB add operation, with or without batch

procedure _MaxShardCount;
Validate TSQLRestStorageShardDB.MaxShardCount implementation

procedure _RTree;
Validate RTREE virtual tables

procedure _TSQLTableWritable;
Test the TSQLTableWritable table
**TTestFileBasedWAL = class(TTestFileBased)**

This test case will test most functions, classes and types defined and implemented in the mORMotSQLite3 unit, i.e. the SQLite3 engine itself, with a file-based approach
- purpose of this class is to test Write-Ahead Logging for the database

Used for DI-2.2.2 (page 2549).

**TTestFileBasedMemoryMap = class(TTestFileBased)**

This test case will test most functions, classes and types defined and implemented in the mORMotSQLite3 unit, i.e. the SQLite3 engine itself, with a file-based approach
- purpose of this class is to test Memory-Mapped I/O for the database

**TTestClientServerAccess = class(TSynTestCase)**

This test case will test most functions, classes and types defined and implemented in the mORMotSQLite3 unit, i.e. the SQLite3 engine itself, used as a HTTP/1.1 server and client
- test a HTTP/1.1 server and client on the port 888 of the local machine
- require the ‘test.db3’ SQLite3 database file, as created by TTestFileBased

Used for DI-2.2.2 (page 2549).

class function RegisterAddUrl(OnlyDelete: boolean): string;

This could be called as administrator for THttpApiServer to work

procedure DirectInProcessAccess;

Validate the client implementation, using direct access to the server
- it connects directly the client to the server, therefore use the same process and memory during the run: it's the fastest possible way of communicating
- it then runs 1000 remote SQL queries, and check the JSON data retrieved
- the time elapsed for this step is computed, and displayed on the report

procedure HTTPClientCustomEncryptionAes;

Validates TSQLRest.SetCustomEncryption process with only AES

procedure HTTPClientCustomEncryptionAesSha;

Validates TSQLRest.SetCustomEncryption process with AES+SHA

procedure HTTPClientCustomEncryptionSha;

Validates TSQLRest.SetCustomEncryption process with only SHA

procedure HTTPClientEncrypted;

Validates the HTTP/1.1 client multi-query implementation with one connection for the all queries and our proprietary SHA-256 / AES-256-CTR encryption encoding
- it runs 1000 remote SQL queries, and check the JSON data retrieved
- the time elapsed for this step is computed, and displayed on the report

procedure HTTPClientKeepAlive;

Validate the HTTP/1.1 client multi-query implementation with one connection for the all queries
- this method keep alive the HTTP connection, so is somewhat faster
- it runs 1000 remote SQL queries, and check the JSON data retrieved
- the time elapsed for this step is computed, and displayed on the report
procedure HTTPClientMultiConnect;
  Validate the HTTP/1.1 client multi-query implementation with one connection initialized per query
  - this method don’t keep alive the HTTP connection, so is somewhat slower: a new HTTP connection is created for every query
  - it runs 1000 remote SQL queries, and check the JSON data retrieved
  - the time elapsed for this step is computed, and displayed on the report

procedure HTTPSeveralDBServers;
  Validate HTTP/1.1 client-server with multiple TSQLite3Server instances

procedure LocalWindowMessages;
  Validate the Windows Messages based client implementation
  - it first launch the Server to handle Windows Messages
  - it then runs 1000 remote SQL queries, and check the JSON data retrieved
  - the time elapsed for this step is computed, and displayed on the report

procedure NamedPipeAccess;
  Validate the Named-Pipe client implementation
  - it first launch the Server as Named-Pipe
  - it then runs 1000 remote SQL queries, and check the JSON data retrieved
  - the time elapsed for this step is computed, and displayed on the report

procedure _TSQLiteHttpClient;
  Validate the HTTP/1.1 client implementation
  - by using a request of all records data

procedure _TSQLiteHttpServer;

  Initialize a TSQLiteHttpServer instance
  - uses the 'test.db3' SQLite3 database file generated by TSQLite3Engine
  - creates and validates a HTTP/1.1 server on the port 888 of the local machine, using the THttpApiServer (using kernel mode http.sys) class if available

TSQLite3ServerTest = class(TSQLite3ServerDB)

  This class defined two published methods of type TSQLite3ServerCallback in order to test the Server-side ModelRoot/TableName/ID/MethodName RESTful model
procedure DataAsHex(Ctxt: TSQLRestServerURIContext);

Test ModelRoot/People/ID/DataAsHex
- this method is called by TSQLRestServer.URI when a ModelRoot/People/ID/DataAsHex GET request is provided
- Parameters values are not used here: this service only need aRecord.ID
- SentData is set with incoming data from a PUT method
- if called from ModelRoot/People/ID/DataAsHex with GET or PUT methods, TSQLRestServer.URI will create a TSQLRecord instance and set its ID (but won't retrieve its other field values automatically)
- if called from ModelRoot/People/DataAsHex with GET or PUT methods, TSQLRestServer.URI will leave aRecord.ID=0 before launching it
- if called from ModelRoot/DataAsHex with GET or PUT methods, TSQLRestServer.URI will leave aRecord=nil before launching it
- implementation must return the HTTP error code (e.g. 200 as success)
- Table is overloaded as TSQLRecordPeople here, and still match the TSQLRestServerCallBack prototype: but you have to check the class at runtime: it can be called by another similar but invalid URL, like ModelRoot/OtherTableName/ID/DataAsHex

procedure Sum(Ctxt: TSQLRestServerURIContext);

Method used to test the Server-Side ModelRoot/Sum or ModelRoot/People/Sum Requests with JSON process
- implementation of this method returns the sum of two floating-points, named A and B, as in the public TSQLRecordPeople.Sum() method, which implements the Client-Side of this service
- Table nor ID are never used here

procedure Sum2(Ctxt: TSQLRestServerURIContext);

Method used to test the Server-Side ModelRoot/Sum or ModelRoot/People/Sum Requests with variant process

TTestExternalDatabase = class(TSynTestCase)

A test case which will test most external DB functions of the mORMotDB unit
- the external DB will be in fact a SynDBSQLite3 instance, expecting a test.db3 SQLite3 file available in the current directory, populated with some TSQLRecordPeople rows
- note that SQL statement caching at SQLite3 engine level makes those test 2 times faster: nice proof of performance improvement

procedure AutoAdaptSQL;

Check the SQL auto-adaptation features

procedure CryptedDatabase;

Check the per-db encryption
- the testpass.db3-wal file is not encrypted, but the main testpass.db3 file will

procedure DBPropertiesPersistence;

Test TSQLDBConnectionProperties persistent as JSON

procedure ExternalRecords;

Initialize needed RESTful client (and server) instances
- i.e. a RESTful direct access to an external DB
procedure ExternalViaREST;
    Test external DB implementation via faster REST calls
    - will mostly call directly the TSQLRestStorageExternal instance, bypassing the Virtual Table mechanism of SQLite3

procedure ExternalViaRESTWithChangeTracking;
    Test external DB implementation via faster REST calls and change tracking
    - a TSQLRecordHistory table will be used to store record history

procedure ExternalViaVirtualTable;
    Test external DB implementation via slower Virtual Table calls
    - using the Virtual Table mechanism of SQLite3 is more than 2 times slower than direct REST access

procedure JETDatabase;
    Test external DB using the JET engine

procedure _SynDBRemote;
    Test SynDB connection remote access via HTTP

procedure _TQuery;
    Test TQuery emulation class

procedure CleanUp; override;
    Release used instances (e.g. server) and memory

TTestMultiThreadProcess = class(TSynTestCase)
A test case for multi-threading abilities of the framework
- will test all direct or remote access protocols with a growing number of concurrent clients (1,2,5,10,30,50 concurrent threads), to ensure stability, scalability and safety of the framework

constructor Create(Owner: TSynTests; const Ident: string = ''); override;
    Create the test case instance

procedure BackgroundThread;
    Test via TSQLRestClientDB instances with AcquireWriteMode=amBackgroundThread

procedure CreateThreadPool;
    Initialize fDatabase and create MaxThreads threads for clients

procedure Locked;
    Test via TSQLRestClientDB instances with AcquireWriteMode=amLocked

procedure MainThread;
    Test via TSQLRestClientDB instances with AcquireWriteMode=amMainThread

procedure SocketAPI;
    Test via TSQLHttpClientWinSock instances over OS's socket API server
    - this test won't work within the Delphi IDE debugger

procedure Unlocked;
    Test via TSQLRestClientDB instances with AcquireWriteMode=amUnlocked
procedure Websockets;
    // test via TSQLHttpclientWebsockets instances

procedure WindowsAPI;
    Test via TSQLHttpclientWinHTTP instances over http.sys (HTTP API) server

procedure _TSQLRestClientDB;
    Test via TSQLRestClientDB instances

procedure _TSQLRestClientURIMessage;
    Test via TSQLRestClientURIMessage instances

procedure _TSQLRestClientURINamedPipe;
    Test via TSQLRestClientURINamedPipe instances

procedure _TSQLRestServerDB;
    Direct test of its RESTful methods

procedure CleanUp; override;
    Release used instances (e.g. server) and memory

property ClientOnlyServerIP: RawByteString read fClientOnlyServerIP write fClientOnlyServerIP;
    If not '', forces the test not to initiate any server and connect to the specified server IP address

property ClientPerThread: Integer read fClientPerThread write fClientPerThread;
    How many TSQLRest instance is initialized per thread
    - is 1 by default

property MaxThreads: integer read fMaxThreads write fMaxThreads;
    The maximum number of threads used for this test
    - is 50 by default

property MinThreads: integer read fMinThreads write fMinThreads;
    The minimum number of threads used for this test
    - is 1 by default

property OperationCount: integer read fOperationCount write fOperationCount;
    How many Add() + Retrieve() operations are performed during each test
    - is 200 by default, i.e. 200 Add() plus 200 Retrieve() globally

IBidirCallback = interface(IInvokable)
    SOA callback definition as expected by TTestBidirectionalRemoteConnection

IBidirService = interface(IInvokable)
    SOA service definition as expected by TTestBidirectionalRemoteConnection

TTestBidirectionalRemoteConnection = class(TSynTestCase)
    A test case for all bidirectional remote access, e.g. WebSockets

procedure RelayConnectionRecreate;
    Verify ability to reconnect from Private Relay to Public Relay
procedure RelayShutdown;
   Finalize SynProtoRelay tunnelling

procedure RelaySOACallbackViaBinaryWebsockets;
   Test SynProtoRelay tunnelling over binary WebSockets

procedure RelaySOACallbackViaJSONWebsockets;
   Test SynProtoRelay tunnelling over JSON WebSockets

procedure RelayStart;
   Initialize SynProtoRelay tunnelling

procedure RunHttpServer;
   Launch the WebSockets-ready HTTP server

procedure SOACallbackOnServerSide;
   Test the callback mechanism via interface-based services on server side

procedure SOACallbackViaBinaryWebsockets;
   Test callbacks via interface-based services over binary WebSockets

procedure SOACallbackViaJSONWebsockets;
   Test callbacks via interface-based services over JSON WebSockets

procedure WebsocketsBinaryProtocol;
   Low-level test of our ‘synopsebinary’ WebSockets binary protocol

procedure WebsocketsJSONProtocol;
   Low-level test of our ‘synopsejson’ WebSockets JSON protocol

procedure _TRecordVersion;
   Test Master/Slave replication using TRecordVersion field over WebSockets

TDDDTest = class(TSynPersistent)
   This is our simple Test data class. Will be mapped to TSQLRecordDDDTest.

TSQLRecordDDDTest = class(TSQLRecord)
   The corresponding TSQLRecord for TDDDTest.

IDDDThreadsQuery = interface(ICQRSService)
   CQRS Query Interface for TTest

IDDDThreadsCommand = interface(IDDDThreadsQuery)
   CQRS Command Interface for TTest

TTestDDDSharedUnits = class(TSynTestCase)
   A test case for all shared DDD types and services

procedure AuthenticationModel;
   Test the Authentication modelization types, and implementation

procedure EmailValidationProcess;
   Test the Email validation process
procedure UserCQRSRepository;
    Test the CQRS Repository for TUser persistence

procedure UserModel;
    Test the User modelization types, including e.g. Address

TTTestDDDMultiThread = class(TSynTestCase)
    A test case for aggressive multi-threaded DDD ORM test

procedure DeleteOldDatabase;
    Delete any old Test database on start

procedure MultiThreadedClientsTest;
    Test concurrent access with multiple clients

procedure SingleClientTest;
    Test straight-forward access using 1 thread and 1 client

procedure StartServer;
    Start the whole DDD Server (http and rest)

TComplexNumber = class(TPersistent)
    A test class, used by TTTestServiceOrientedArchitecture
    - to test TPersistent objects used as parameters for remote service calls

constructor Create(aReal, aImaginary: double); reintroduce;
    Create an instance to store a complex number

property Imaginary: Double read fImaginary write fImaginary;
    The imaginary part of this complex number

property Real: Double read fReal write fReal;
    The real part of this complex number

TConsultaNav = packed record
    A record used by IComplexCalculator.EchoRecord

TCustomerData = packed record
    A record used by IComplexCalculator.GetCustomer

ICalculator = interface(IInvokable)
    A test interface, used by TTTestServiceOrientedArchitecture
    - to test basic and high-level remote service calls

function Add(n1,n2: integer): integer;
    Add two signed 32 bit integers
function ComplexCall(const Ints: TIntegerDynArray; const Strs1: TRawUTF8DynArray; var Str2: TWideStringDynArray; const Rec1: TVirtualTableModuleProperties; var Rec2: TSQLRestCacheEntryValue; float1: double; var float2: double):
  TSQLRestCacheEntryValue;
  Test integer, strings and wide strings dynamic arrays, together with records

function DirectCall(const Data: TSQLRawBlob): integer;
  Validates ArgsInPutIsOctetStream raw binary upload

function Multiply(n1, n2: Int64): Int64;
  Multiply two signed 64 bit integers

function RepeatJsonArray(const item: RawUTF8; count: integer): RawJSON;
  Validates huge RawJSON/RawUTF8

function SpecialCall(Txt: RawUTF8; var Int: integer; var Card: cardinal; field: TSynTableFieldTypes; fields: TSynTableFieldTypes; var options: TSynTableFieldOptions): TSynTableFieldTypes;
  Do some work with strings, sets and enumerates parameters, testing also var (in/out) parameters and set as a function result

function StackFloatMultiply(n1, n2, n3, n4, n5, n6, n7, n8, n9, n10: double): Int64;
  Test float stack access

function StackIntMultiply(n1, n2, n3, n4, n5, n6, n7, n8, n9, n10: integer): Int64;
  Test unaligned stack access

function Subtract(n1, n2: double): double;
  Subtract two floating-point values

function ToTextFunc(Value: double): string;
  Convert a floating-point value into text

procedure Swap(var n1, n2: double);
  Swap two by-reference floating-point values
  - would validate pointer use instead of XMM1/XMM2 registers under Win64

procedure ToText(Value: Currency; var Result: RawUTF8);
  Convert a currency value into text

IComplexCalculator = interface(ICalculator)
A test interface, used by TTestServiceOrientedArchitecture
- to test remote service calls with objects as parameters (its published properties will be serialized as standard JSON objects)
- since it inherits from ICalculator interface, it will also test the proper interface inheritance handling (i.e. it will test that ICalculator methods are also available)

function GetCurrentThreadID: PtrUInt;
  Returns the thread ID running the method on server side

function GetCustomer(CustomerId: Integer; out CustomerData: TCustomerData): Boolean;
  Validate record transmission
function IsNull(n: TComplexNumber): boolean;
    Purpose of this method is to check for boolean handling.

function TestBlob(n: TComplexNumber): TServiceCustomAnswer;
    This will test the BLOB kind of remote answer.

function TestVariants(const Text: RawUTF8; V1: Variant; var V2: variant): variant;
    Test variant kind of parameters.

procedure Collections(Item: TCollTest; var List: TCollTestsI; out Copy: TCollTestsI);
    Test in/out collections.

procedure FillPeople(var People: TSQLRecordPeople);
    / validate TSQLRecord transmission

procedure Subtract(n1,n2: TComplexNumber; out Result: TComplexNumber);
    Purpose of this method is to subtract two complex numbers
    - using class instances as parameters.

IComplexNumber = interface(IInvokable)
    A test interface, used by TTestServiceOrientedArchitecture
    - to test sicClientDriven implementation pattern: data will remain on the server until the
    IComplexNumber instance is out of scope.

ITestUser = interface(IInvokable)
    A test interface, used by TTestServiceOrientedArchitecture
    - to test sicPerUser implementation pattern.

ITestGroup = interface(ITestUser)
    A test interface, used by TTestServiceOrientedArchitecture
    - to test sicPerGroup implementation pattern.

ITestSession = interface(ITestUser)
    A test interface, used by TTestServiceOrientedArchitecture
    - to test sicPerSession implementation pattern.

ITestPerThread = interface(IInvokable)
    A test interface, used by TTestServiceOrientedArchitecture
    - to test threading implementation pattern.

TUser = record
    A test value object, used by IUserRepository/ISmsSender interfaces
    - to test stubing/mocking implementation pattern.

IUserRepository = interface(IInvokable)
    A test interface, used by TTestServiceOrientedArchitecture
    - to test stubing/mocking implementation pattern.

ISmsSender = interface(IInvokable)
    A test interface, used by TTestServiceOrientedArchitecture
    - to test stubing/mocking implementation pattern.
TTestServiceOrientedArchitecture = class(TSynTestCase)
  A test case which will test the interface-based SOA implementation of the mORMot framework

procedure ClientSideJSONRPC;
  Test the client-side implementation in JSON-RPC mode

procedure ClientSideREST;
  Test the client-side implementation in RESTful mode

procedure ClientSideRESTAsJSONObject;
  Test the client-side implementation in RESTful mode with values transmitted as JSON objects

procedure ClientSideRESTBackgroundThread;
  Test the client-side implementation of opt*InPerInterfaceThread option

procedure ClientSideRESTBasicAuthentication;
  Test the client-side implementation using TSQLRestServerAuthenticationHttpBasic

procedure ClientSideRESTCustomRecordLayout;
  Test the custom record JSON serialization

procedure ClientSideRESTLocked;
  Test the client-side implementation of optExecLockedPerInterface

procedure ClientSideRESTMainThread;
  Test the client-side implementation of opt*InMainThread option

procedure ClientSideRESTServiceLogToDB;
  Test the client-side in RESTful mode with all calls logged in a table

procedure ClientSideRESTSessionsStats;
  Test the client-side in RESTful mode with full session statistics

procedure ClientSideRESTSignWithCrc32c;
  Test the client-side implementation with crc32c URI signature

procedure ClientSideRESTSignWithMd5;
  Test the client-side implementation with MD5 URI signature

procedure ClientSideRESTSignWithSha256;
  Test the client-side implementation with SHA256 URI signature

procedure ClientSideRESTSignWithSha512;
  Test the client-side implementation with SHA512 URI signature

procedure ClientSideRESTSignWithXxhash;
  Test the client-side implementation with xxHash32 URI signature

procedure ClientSideRESTWeakAuthentication;
  Test the client-side implementation using TSQLRestServerAuthenticationNone

procedure DirectCall;
  Test direct call to the class instance
procedure MocksAndStubs;
   Test interface stubbing / mocking

procedure Security;
   Test the security features

procedure ServerSide;
   Test the server-side implementation

procedure ServiceInitialization;
   Initialize the SOA implementation

procedure TestOverHTTP;
   Test RESTful mode using HTTP client/server communication

procedure WeakInterfaces;
   Test the SetWeak/SetWeakZero weak interface functions

Constants implemented in the SynSelfTests unit

FIREBIRDEMBEDDEDDLL = 'd:\Dev\Lib\SQLite3\Samples\15 - External DB performance\Firebird'+ HTTP_DEFAULTPORT = '8888';

download driver from http://www.firebirdsql.org/en/odbc-driver
27.33. SynSM.pas unit

*Purpose*: Features JavaScript execution using the SpiderMonkey library
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynSM unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1363</td>
</tr>
<tr>
<td>SynSMAPI</td>
<td>SpiderMonkey *.h header port to Delphi - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1573</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features - as a complement to SynCommons, which tended to increase too much - licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1721</td>
</tr>
<tr>
<td>SynTests</td>
<td>Unit test functions used by Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1832</td>
</tr>
</tbody>
</table>

**SynSM class hierarchy**

**Objects implemented in the SynSM unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynSM.pas unit - Rev. 1.18</td>
<td>Page 1555 of 2552</td>
<td></td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>ESMException</td>
<td>Generic parent class of all SpiderMonkey-related Exception types</td>
<td>1555</td>
</tr>
<tr>
<td>TSMEngine</td>
<td>Implements a ThreadSafe JavaScript engine</td>
<td>1564</td>
</tr>
<tr>
<td>TSMEngineManager</td>
<td>Main access point to the SpiderMonkey per-thread scripting engines</td>
<td>1567</td>
</tr>
<tr>
<td>TSMEngineMethodEvent</td>
<td>Used to store one registered method event</td>
<td>1563</td>
</tr>
<tr>
<td>TSMObject</td>
<td>Just a wrapper around JavaScript Object API type, to be used with other</td>
<td>1559</td>
</tr>
<tr>
<td></td>
<td>values wrappers</td>
<td></td>
</tr>
<tr>
<td>TSMValue</td>
<td>Just a wrapper around jsval API type, to be used with our object wrappers</td>
<td>1555</td>
</tr>
<tr>
<td>TSMVariant</td>
<td>A custom variant type used to store a SpiderMonkey object in Delphi code</td>
<td>1568</td>
</tr>
<tr>
<td>TSMVariantData</td>
<td>Memory structure used for TSMVariant storage of any JavaScript object as</td>
<td>1569</td>
</tr>
<tr>
<td></td>
<td>Delphi variant</td>
<td></td>
</tr>
</tbody>
</table>

ESMException = class(ESynException)

Generic parent class of all SpiderMonkey-related Exception types

TSMValue = object(TObject)

Just a wrapper around jsval API type, to be used with our object wrappers
- SpiderMonkey jsval type can be directly casted to this type via TSMValue(jsval)
- note that some methods expect an execution context to be supplied as parameter, as soon as it contains a non primitive type (double/integer)

function SetJSON(cx: PJSContext; const aJSON: RawUTF8): boolean;
Set the value from UTF-8 encoded JSON
- returns TRUE if aJSON was valid, FALSE in case of an error

function ToBoolean: boolean;
Read the value as boolean

function ToDateTime(cx: PJSContext): TDateTime;
Return the value as a date/time
- in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know the execution context (using a property is not an option)

function ToDouble: double;
Read the value as floating point

function ToInt64: int64;
Read the value as one 64 bit integer
- note that SpiderMonkey is not able to store all Int64 values directly

function ToInteger: integer;
Read the value as one 32 bit integer
function ToJSON(cx: PJSContext): RawUTF8;
  Return the value as UTF-8 encoded JSON

function ToNativeFunction(cx: PJSContext): PJSSFunction;
  Attempts to convert the value into a native function pointer

function ToNativeFunctionName(cx: PJSContext): RawUTF8;
  Attempts to convert the value into a native function name

function ToSynUnicode(cx: PJSContext): SynUnicode; overload;
  Return the value as an Unicode String
  - in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know
    the execution context (using a property is not an option)

function ToUTF8(cx: PJSContext): RawUTF8;
  Return the value as an UTF-8 String
  - in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know
    the execution context (using a property is not an option)

function ToVariant(cx: PJSContext): Variant; overload;
  Return the value as variant (not implemented yet)
  - will return any JavaScript string value directly as a RawUTF8
  - will return any JavaScript object value as a TDocVariant document
  - in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know
    the execution context (using a property is not an option)

function ToWideString(cx: PJSContext): WideString;
  Return the value as an Unicode WideString
  - in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know
    the execution context (using a property is not an option)

function TransformToSynUnicode(cx: PJSContext): SynUnicode;
  Transform a JSValue to its UTF-16 string representation
  - JavaScript equivalent is variable.toString()

function TransformToUTF8(cx: PJSContext): RawUTF8;
  Transform a JSValue to its UTF-8 string representation
  - JavaScript equivalent is variable.toString()

function ValType(cx: PJSContext): JSType;
  Type of the value
  - you should better use this before calling other To*() methods

procedure AddJSON(cx: PJSContext; W: TTextWriter);
  Add the value as UTF-8 encoded JSON
procedure SetAnsiChar(cx: PJSContext; Text: PAnsiChar; TextLen, CodePage: integer);
   Set the value as an Ansi encoded buffer (may be UTF-8 or any code page)
   - if CodePage is 0, will use the CurrentAnsiCodePage value
   - in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know
     the execution context (using a property is not an option)
   - warning - JSString string is a subject for GC so you must root it or set as property of some
     object or use SetNativeString() method to pass the value by reference

procedure SetBoolean(const Value: boolean);
   Set the value as boolean

procedure SetDateTime(cx: PJSContext; const Value: TDateTime);
   Set the value as a date/time
   - in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know
     the execution context (using a property is not an option)

procedure SetDouble(const Value: double);
   Set the value as floating point

procedure SetInt64(const Value: int64);
   Set the value as one 64 bit integer
   - this is a somewhat dirty hack, since SpiderMonkey don't support int64: but it is possible to
     transform int64 to double for ant value < (1 shl 51)
   - sometimes we need int64 to be passed do SpiderMonkey (e.g. for an ID)

procedure SetInteger(const Value: integer);
   Set the value as one 32 bit integer

procedure SetNativeString(cx: PJSContext; const aStr: SynUnicode);
   Set the value as Unicode String by reference
   - this is the fastest way to add a string to SpiderMonkey: String is in fact not copied to the
     SpiderMonkey engine, just passed by reference
   - Only SynUnicode string support by now (SpiderMonkey is internally UTF-16 based)
   - WARNING - as a consequence, aStr must be UNCHANGED until SpiderMonkey engine points to
     it (SpiderMonkey will also consider its strings as immutable, so will never change its content
     during execution) - for instance, never pass a function result as aStr, nor use a local SynUnicode
     variable unless you trigger the Garbage Collection before the end of the local method

procedure SetNull;
   Set the value as NULL

procedure SetSynUnicode(cx: PJSContext; const aStr: SynUnicode);
   Set the value as an Unicode String
   - in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know
     the execution context (using a property is not an option)
   - warning - JSString string is a subject for GC so you must root it or set as property of some
     object or use SetNativeString() method to pass the value by reference
procedure SetTVarRec(cx: PJSContext; const V: TVarRec);

Set the value as TVarRec (i.e. an "array of const" open parameter)
- here any AnsiString parameter is expected to be a RawUTF8 before Delphi 2009, or its correct code page will be retrieved since Delphi 2009
- in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know the execution context (using a property is not an option)

procedure SetUTF8(cx: PJSContext; const aStr: RawUTF8);

Set the value as an UTF-8 String
- in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know the execution context (using a property is not an option)
- warning - JSString string is a subject for GC so you must root it or set as property of some object or use SetNativeString() method to pass the value by reference

procedure SetVariant(cx: PJSContext; const Value: Variant);

Set the value as variant (not implemented yet)
- will set any custom variant type (e.g. TDocVariant) as a JavaScript object value computed from the JSON serialization of the variant
- in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know the execution context (using a property is not an option)

procedure SetVoid;

Set the value as VOID

procedure SetWideChar(cx: PJSContext; Text: PWideChar; TextLen: integer);

Set the value as an UTF-16 encoded buffer
- in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know the execution context (using a property is not an option)
- warning - JSString string is a subject for GC so you must root it or set as property of some object or use SetNativeString() method to pass the value by reference

procedure Set WideString(cx: PJSContext; const aStr: WideString);

Set the value as an Unicode WideString
- in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know the execution context (using a property is not an option)
- warning - JSString string is a subject for GC so you must root it or set as property of some object or use SetNativeString() method to pass the value by reference

procedure ToSynUnicode(cx: PJSContext; var result: SynUnicode); overload;

Return the value as an Unicode String
- in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know the execution context (using a property is not an option)

procedure ToVariant(cx: PJSContext; var result: Variant); overload;

Return the value as variant (not implemented yet)
- will return any JavaScript string value directly as a RawUTF8
- will return any JavaScript object value as a TDocVariant document
- in SpiderMonkey non-simple type instances do exist in a given JSContext, so we need to know the execution context (using a property is not an option)

property AsBoolean: boolean read ToBoolean write SetBoolean;

Access to the value as boolean
property AsDouble: double read ToDouble write SetDouble;
  Access to the value as floating point

property AsInt64: int64 read ToInt64 write SetInt64;
  Access to the value as one 64 bit integer

property AsInteger: integer read ToInteger write SetInteger;
  Access to the value as integer

property AsJSVal: jsval read FValue write FValue;
  Direct access to the internal jsval instance

TSMObject = object(TObject)
  Just a wrapper around JavaScript Object API type, to be used with other values wrappers
  - SpiderMonkey object type can NOT be directly casted to this type via TSMObject(jsobject) - use
    JSObject wrapper instead - since we expects an execution context to be specified
  - to create instance of this structure, use TSMEngine.NewObject() or MakeObject() overloaded
    methods

function AsSMValue: TSMValue;
  Returns the associated jsobject instance as a jsvalue

function DefineNativeMethod(const methodName: SynUnicode; func: JSNative; nargs: uintN; attrs: TJSPropertyAttrs): PJSFunction; overload;
  Add JSNative compatible function into JS object
  - here the method name is specified as SynUnicode
  - func if reference to function with JSNative signature
  - nargs is function argument count
  - actually this method creates a JSFunction and assing its value to
    obj[methodName]
  - to add a global function, define it into the "global" object - i.e. call
    TSMEngine.GlobalObject.DefineNativeMethod()
  - this method will allow to set custom properties attributes of this engine

function DefineNativeMethod(const methodName: SynUnicode; func: JSNative; nargs: uintN): PJSFunction; overload;
  Add JSNative compatible function into JS object
  - here the method name is specified as SynUnicode
  - func if reference to function with JSNative signature
  - nargs is function argument count
  - actually this method creates a JSFunction and assing its value to
    obj[methodName]
  - to add a global function, define it into the "global" object - i.e. call
    TSMEngine.GlobalObject.DefineNativeMethod()
  - this method will use the default properties attributes of this engine
function DefineNativeMethod(const methodName: AnsiString; func: JSNative; nargs: uintN; attrs: TJSPropertyAttrs): PJSFunction; overload;

Add JSNative compatible function into JS object
- here the method name is specified as AnsiString
- func if reference to function with JSNative signature
- nargs is function argument count
- this method will allow to set custom properties attributes of this engine

function DefineNativeMethod(const methodName: AnsiString; func: JSNative; nargs: uintN): PJSFunction; overload;

Add JSNative compatible function into JS object
- here the method name is specified as AnsiString
- func if reference to function with JSNative signature
- nargs is function argument count
- this method will use the default properties attributes of this engine

function Engine: TSMEngine;

Returns the associated script engine instance

function GetPrivateData(expectedClass: PJSClass): pointer;

Retrieve the private data associated with an object, if that object is an instance of a specified class
- wrapper to JS_GetInstancePrivate()

function GetPropValue(const propName: SynUnicode): TSMValue;

Get object property value (call getter for native)
- JavaScript equivalent of obj[name]
- returns JSVAL_VOID if object does not have such property

function GetPropVariant(const propName: SynUnicode): variant;

Get object property value (call getter for native)
- you can also use the property Properties[]
- JavaScript equivalent of obj[name]
- returns null if object does not have such property

function HasOwnProperty(const propName: SynUnicode): Boolean;

Determine whether a property is physically present on a object
- JavaScript equivalent of Object.hasOwnProperty(propName)

function HasProperty(const propName: SynUnicode): Boolean;

Check object property does exist (including prototype chain lookup)

function IsArray: boolean;

Return TRUE if the object is an array

function ItemsCount: cardinal;

Return the number of elements in this array
**function** Parent: TSMObject;
*Get the parent object of a given object*

**function** Prototype: TSMObject;
*Get the prototype of a given object*

**function** Run(const methodName: AnsiString; const argv: array of variant): variant;
*Executes a JavaScript object method using a Delphi array of variants*
- returns the function result as a variant
- JavaScript equivalent of
  
rval := obj.methodName(argv[0], ....);

**procedure** Clear;
*Set properties obj and cx to nil*

**procedure** DefineProperty(const name: SynUnicode; const value: TSMValue); overload;
*Define an object property with a value, specified as jsvalue*
- this is not a direct JavaScript equivalent of
  
  obj[name] = val

  since any setter will be called
- to set a property in a global object, call either
  
  SMEngine.Global.property := ...  // via late-binding
  SMEngine.GlobalObject.DefineProperty()  // direct via TSMObject

  equivalent in JavaScript to:
  
  var name = value

  outside a JavaScript function context (i.e. in global scope)
  - if property already exists, it will just replace its value with the supplied value
  - this method will use the default properties attributes of this engine

**procedure** DefineProperty(const name: SynUnicode; const value: variant); overload;
*Define an object property with a value, specified as variant*
- you can also use the property Properties[]
- this is not a direct JavaScript equivalent of
  
  obj[name] = val

  since any setter will be called
- to set a property in a global object, call either
  
  SMEngine.Global.property := ...  // via late-binding
  SMEngine.GlobalObject.DefineProperty()  // direct via TSMObject

  equivalent in JavaScript to:
  
  var name = value

  outside a JavaScript function context (i.e. in global scope)
  - if property already exists, it will just replace its value with the supplied value
  - this method will use the default properties attributes of this engine
**procedure** DefineProperty(const name: SynUnicode; const value: variant; attrs: TJSPropertyAttrs); overload;

*Define an object property with a value, specified as variant*
- you can also use the property Properties[]
- this is not a direct JavaScript equivalent of
  \[obj[name] = val\]
  since any setter will be called
- to set a property in a global object, call either
  \[
  \text{SMEngine.Global.}property := ... // via late-binding
  \text{SMEngine.GlobalObject.}DefineProperty() // direct via TSMObject\]

  equivalent in JavaScript to:
  \[
  \text{var name = value}
  \]
  outside a JavaScript function context (i.e. in global scope)
  - if property already exists, it will just replace its value with the supplied value
  - this method will allow to set custom properties attributes of this engine

**procedure** DefineProperty(const name: SynUnicode; const value: TSMValue; attrs: TJSPropertyAttrs); overload;

*Define an object property with a value, specified as jsvalue*
- this is not a direct JavaScript equivalent of
  \[obj[name] = val\]
  since any setter will be called
- to set a property in a global object, call either
  \[
  \text{SMEngine.Global.}property := ... // via late-binding
  \text{SMEngine.GlobalObject.}DefineProperty() // direct via TSMObject\]

  equivalent in JavaScript to:
  \[
  \text{var name = value}
  \]
  outside a JavaScript function context (i.e. in global scope)
  - if property already exists, it will just replace its value with the supplied value
  - this method will allow to set custom properties attributes of this engine

**procedure** DeleteItem(aIndex: integer);

*Delete an item of this object as array*

**procedure** Evaluate(const script: SynUnicode; const scriptName: RawUTF8; lineNo: Cardinal; out result: TSMValue);

*Evaluate JavaScript script in the current object scope*
- if exception raised in script - raise Delphi ESMException
- on success, returns the last executed expression statement processed in the script in low-level result output variable
- JavaScript Equivalent of
  \[
  \text{with(obj) eval(script)}
  \]
  - be careful about execution scope - see JS_ExecuteScript() description
  - usualy you need to evaluate script only in global object scope, so you should better always call
  TSMEngine.Evaluate()

**procedure** Root;

*Protect the object from Garbage Collection*
- if this object is not set as property value of any other object or passed as parameter to function, you must protect it
procedure RunMethod(const methodName: AnsiString; const argv: SMValArray; out rval: TSMValue); overload;
  Executes a JavaScript object method using low-level SMVal arguments
  - returns the function result as a TSMValue
  - JavaScript equivalent of
    rval := obj.methodName(argv[0], ....);

procedure RunMethod(const methodName: AnsiString; const argv: array of const; out rval: TSMValue); overload;
  Executes a JavaScript object method using a Delphi array of const
  - returns the function result as a TSMValue
  - JavaScript equivalent of
    rval := obj.methodName(argv[0], ....);
  - here any AnsiString parameter is expected to be a RawUTF8 before Delphi 2009, or its correct code page will be retrieved since Delphi 2009

procedure UnRoot;
  Unprotect a previously "rooted" object
  - WARNING!! Object MUST be protected by a previous Root method call, otherwise you get an access violation

property cx: PJSContext read fCx;
  Returns the associated execution context

property DefaultPropertyAttrs: TJSPropertyAttrs read FDefaultPropertyAttrs write SetDefaultPropertyAttrs;
  Access to the default attributes when accessing any properties

property Items[aIndex: integer]: variant read GetItem write SetItem;
  Access to an item of this object as array

property obj: PJSObject read fObj;
  Returns the associated jsobject instance

property PrivateData: pointer read GetPrivate write SetPrivate;
  Access the private data field of an object
  - wrapper to JS_GetPrivate()/JS_SetPrivate()
  - only works if the object's JSClass has the JSCLASS_HAS_PRIVATE flag: it is safer to use GetPrivateData() method and providing the JSClass

property Properties[const propName: SynUnicode]: variant read GetPropVariant write SetPropVariant;
  Read/write access to the object properties as variant

TSMEngineMethodEvent = record
  Used to store one registered method event
TSMEngine = class(TObject)

Implements a ThreadSafe JavaScript engine
- use TSMEngineManager.ThreadSafeEngine to retrieve the Engine instance corresponding to the current thread, in multithread application
- contains JSRuntime + JSContext (to be ready for new SpiderMonkey version where context and runtime is the same)
- contains also one "global" JavaScript object. From script it is accessible via "global." (in browser, this is the "window." object)
- set SpiderMonkey error reporter and store last SpiderMonkey error in LastError property

constructor Create(aManager: TSMEngineManager); virtual;
Create one threadsafe JavaScript Engine instance
- initialize internal JSRuntime, JSContext, and global objects and standard JavaScript classes
- do not create Engine directly via this constructor, but instead call TSMEngineManager.ThreadSafeEngine

destructor Destroy; override;
Finalize the JavaScript engine instance

function Evaluate(const script: SyncUnicode; const scriptName: RawUTF8='script'; lineNo: Cardinal=1): variant;
Evaluate a JavaScript script in the global scope
- a wrapper to GlobalObject.Eval
- if exception raised in script - raise Delphi ESMException
- on success returns last executed expression statement processed in the script as a variant
- JavaScript equivalent to eval(script)

function NewSMVariant: variant;
Create new ordinary JavaScript object, stored as TSMVariant custom type
- JavaScript equivalent of {}
- new object is subject to Garbage Collection, so should be assigned as value for a property to create new object type property, as in JavaScript:
  var obj = {}
function RegisterMethod(obj: PJSObject; const MethodName: SynUnicode; const Event: TSMEngineMethodEventVariant; ArgumentsCount: integer): PJSFunction; overload;

Register a native Delphi variant-based method for a given object
- the supplied function name is case-sensitive
- the supplied callback will be executed directly by the JavaScript engine, supplying all parameters as variant (including TDocVariant for any complex object), and returning the function result as variant
- raise an ESMException if the function could not be registered

procedure CheckJSError(res: JSBool); virtual;
Check if last call to JSAPI compile/eval function was successful
- raise ESMException if any error occurred
- put error description to SynSMLog

procedure ClearLastError;
Clear last JavaScript error
- called before every evaluate() function call

procedure GarbageCollect;
Trigger Garbage Collection
- all unrooted things (JSString, JSObject, VVal) will be released

procedure InitClass(clasp: PJSClass; ps: PJSPropertySpec; var newobj: TSMObject);
Create new JavaScript object from its class and property specifications

procedure MakeObject(const value: TSMValue; out obj: TSMObject); overload;
Converts a JavaScript value into a JavaScript object

procedure MakeObject(const value: jsv; out obj: TSMObject); overload;
Converts a JavaScript low-level value into a JavaScript object

procedure MakeObject(jsobj: PJSObject; out obj: TSMObject); overload;
Converts a JavaScript low-level object into a JavaScript object

procedure MaybeGarbageCollect;
Offer the JavaScript engine an opportunity to perform garbage collection if needed
- Tries to determine whether garbage collection in would free up enough memory to be worth the amount of time it would take. If so, it performs some garbage collection
- Frequent calls are safe and will not cause the application to spend a lot of time doing redundant garbage collection work

procedure NewObject(const prototype: TSMObject; out newobj: TSMObject); overload;
Create new JavaScript object with prototype
- JavaScript equivalent of
  
    {}.__proto__ = prototype;
procedure NewObject(out newobj: TSMObject); overload;

Create new ordinary JavaScript object
- JavaScript equivalent of
  
  `{}`
- new object is subject to Garbage Collection, so must be rooted or assigned as value for a property to create new object type property, as in JavaScript:
  
  ```
  var obj = {};
  ```

procedure NewObjectWithClass(clasp: PJSClass; const prototype: TSMObject; const parent: TSMObject; var newobj: TSMObject); overload;

Create new JavaScript object from its prototype

procedure NewObjectWithClass(clasp: PJSClass; var newobj: TSMObject); overload;

Create new JavaScript object from its class

procedure NewSMVariantRooted(out newobj: variant);

Create new ordinary JavaScript object, stored as TSMVariant custom type, and rooted to avoid garbage collection
- JavaScript equivalent of
  
  `{}`
- new object is subject to Garbage Collection, so is rooted and should be explicitly unrooted, e.g.
  
  ```
  obj: variant;
  ...
  FManager.ThreadSafeEngine.NewSMVariantRooted(obj);
  try
    ... work with obj
  finally
    obj._UnRoot; // pseudo-method
  end;
  ```

procedure UnRegisterMethod(JSFunction: PJSFunction);

Unregister a native Delphi method for a given object
- raise an ESMException if the function was not previously registered
- you should not call it usually, but it is available in case

property comp: PJSCompartment read fcomp;

Access to the associated execution compartment

property cx: PJSContext read fCx;

Access to the associated execution context

property DefaultPropertyAttrs: TJSPROPERTYATRROPS read FDefaultPropertyAttrs write SetDefaultPropertyAttrs;

Access to the default attributes when accessing any properties

property EngineContentVersion: Cardinal read FEngineContentVersion;

Internal version number of engine scripts
- used in TSMEngine.ThreadSafeEngine to determine if context is up to date, in order to trigger on-the-fly reload of scripts without the need if restarting the application
- caller must change this parameter value e.g. in case of changes in the scripts folder in an HTTP server
property ErrorExist: boolean read FErrorExist;
    TRUE if an error was triggered during JavaScript execution

property Global: variant read FGlobal;
    Access to the associated global object as a TSMVariant custom variant
    - allows direct property and method executions in Delphi code, via late-binding, for instance:
      engine.Global.MyFunction(1,'text');

property GlobalObj: PJSObject read FGlobalObject.fobj;
    Access to the associated global object as low-level PJSObject

property GlobalObject: TSMObject read FGlobalObject;
    Access to the associated global object as a TSMObject wrapper
    - you can use it to register a method

property LastErrorFileName: RawUTF8 read FLastErrorFileName;
    Last error file name triggered during JavaScript execution

property LastErrorLine: integer read FLastErrorLine;
    Last error source code line number triggered during JavaScript execution

property LastErrorMsg: RawUTF8 read FLastErrorMsg;
    Last error message triggered during JavaScript execution

property rt: PJSRuntime read frt;
    Access to the associated execution runtime

property TimeOutAborted: boolean read FTimeOutAborted;
    Notifies a WatchDog timeout

property TimeOutValue: Double read fTimeOutInterval write SetTimeoutValue;
    Define a WatchDog timeout interval
    - is set to -1 by default, i.e. meaning no execution timeout

TSMEngineManager = class(TObject)
    Main access point to the SpiderMonkey per-thread scripting engines
    - allow thread-safe access to an internal per-thread TSMEngine instance list
    - contains runtime-level properties shared between thread-safe engines
    - you can create several TSMEngineManager instances, if you need several separate scripting instances
    - set OnNewEngine callback to initialize each TSMEngine, when a new thread is accessed, and tune per-engine memory allocation via MaxPerEngineMemory and MaxRecursionDepth
    - get the current per-thread TSMEngine instance via ThreadSafeEngine method

constructor Create; virtual;
    Initialize the SpiderMonkey scripting engine

destructor Destroy; override;
    Finalize the SpiderMonkey scripting engine
**function** ThreadSafeEngine: TSMEngine;

*Get or create one Engine associated with current running thread*
- in single thread application will return the MainEngine

**procedure** ReleaseCurrentThreadEngine;

*Method to be called when a thread is about to be finished*
- you can call this method just before a thread is finished to ensure that the associated scripting Engine will be released
- could be used e.g. in a try...finally block inside a TThread.Execute overridden method

**property** ContentVersion: Cardinal read FContentVersion write FContentVersion;

*Internal version of the script files*
- used in TSMEngine.ThreadSafeEngine to determine if context is up to date, in order to trigger on-the-fly reload of scripts without the need if restarting the application

**property** Lock: TRTLCriticalSection read FEngineCS;

*Lock/mutex used for thread-safe access to the TSMEngine list*

**property** MaxPerEngineMemory: Cardinal read FMaxPerEngineMemory write SetMaxPerEngineMemory default 8*1024*1024;

*Max amount of memory (in bytes) for a single SpiderMonkey instance*
- this parameter will be set only at Engine start, i.e. it must be set BEFORE any call to ThreadSafeEngine
- default is 8 MB

**property** MaxRecursionDepth: Cardinal read FMaxRecursionDepth write FMaxRecursionDepth default 32;

*Maximum expected recursion depth for JavaScript functions*
- to avoid out of memory situation in functions like

  ```plaintext
default 32;  
function f(){ f(); );
```
- default is 32, but you can specify some higher value

**property** OnNewEngine: TEngineEvent read FOnNewEngine write FOnNewEngine;

*Event triggered every time a new Engine is created*
- here your code can change the initial state of the Engine

---

**TSMVariant = class(TSynInvokeableVariantType)**

*A custom variant type used to store a SpiderMonkey object in Delphi code*
- via the magic of late binding, it will allow access of any JavaScript object property, or execute any of its methods
- primitive types (i.e. null, string, or numbers) will be stored as simple variant instances, but
JavaScript objects (i.e. objects, prototypes or functions) can be stored as an instance of this TSMVariant custom type
- you can use the _Root and _UnRoot pseudo-methods, which will protect the object instance to avoid unexpected Garbage Collection

**function** DoFunction(var Dest: TVarData; const V: TVarData; const Name: string; const Arguments: TVarDataArray): Boolean; override;

*Low-level callback to execute any JavaScript object method*
- add the _(Index: integer): variant method to retrieve an item if the object is an array
procedure Cast(var Dest: TVarData; const Source: TVarData); override;
  
Handle type conversion
- any TSMVariant will be converted to '<<JavaScript TSMVariant>>' text

procedure CastTo(var Dest: TVarData; const Source: TVarData; const AVarType: TVarType); override;
  
Handle type conversion
- any TSMVariant will be converted to '<<JavaScript TSMVariant>>' text

class procedure New(const aObject: TSMObject; out aValue: variant); overload;
  
Initialize a variant instance to store a JavaScript object

class procedure New(cx: PJSContext; obj: PJSObject; out aValue: variant); overload;
  
Initialize a variant instance to store a JavaScript object

class procedure New(engine: TSMEngine; out aValue: variant); overload;
  
Initialize a variant instance to store a new JavaScript object

procedure ToJSON(W: TTextWriter; const Value: variant; Escape: TTextWriterKind); override;
  
This implementation will let SpiderMonkey write directly the JSON content

TSMVariantData = object(TObject)
  
Memory structure used for TSMVariant storage of any JavaScript object as Delphi variant
- primitive types (i.e. null, string, or numbers) will be stored as simple variant instances, but
  JavaScript objects (i.e. objects, prototypes or functions) can be stored as an instance of this
  TSMVariant custom type
- this variant stores its execution context, so is pretty convenient to work with in plain Delphi
  code, also thanks to late-binding feature

procedure GetGlobal(out global: variant);
  
Retrieve the global object of this execution context
- you can use this from a native function, e.g.:
  
function TMyClass.MyFunction(const This: variant; const Args: array of variant): variant;
var global: variant;
begin
  TSMVariantData(This).GetGlobal(global);
  global.anotherFunction(Args[0],Args[1], 'test');
  // same as:
  global := TSMVariantData(This).SMObject.Engine.Global;
  global.anotherFunction(Args[0],Args[1], 'test');
  // but you may also write directly:
  with TSMVariantData(This).SMObject.Engine do
  global.anotherFunction(Args[0],Args[1], 'test');
  result := AnyTextFileToSynUnicode(Args[0]);
end;

procedure Init(aCx: PJSContext; aObj: PJSObject); overload;
  
Initialize a TSMVariant structure to store a specified JavaScript object

procedure Init(const aObject: TSMObject); overload;
  
Initialize a TSMVariant structure to store a specified JavaScript object
procedure InitNew(engine: TSMEngine);
  Initialize a TSMVariant structure to store a new JavaScript object

property cx: PJSCxPointer read VObject.fcx;
  Returns the associated execution context

property obj: PJSObject read VObject.fobj;
  Returns the associated jsobject instance

property SMObject: TSMObject read VObject;
  Returns the associated TSMObject instance

property VarType: word read VType;
  Return the custom variant type identifier, i.e. SMVariantType.VarType

Types implemented in the SynSM unit

PSMObject = ^TSMObject;
  Pointer to our wrapper around JavaScript Object

PSMValue = ^TSMValue;
  A pointer to a jsval wrapper

PSMValues = ^TSMValues;
  A pointer to a jsval wrappers array

PSMVariantData = ^TSMVariantData;
  Pointer to a TSMVariant storage

SMValArray = array of TSMValue;
  A dynamic array of jsval wrappers

TEngineEvent = procedure(const Engine: TSMEngine) of object;
  Prototype of SpideMonkey notification callback method

TSMEngineMethodEventDynArray = array of TSMEngineMethodEvent;
  Used to store the registered method events

TSMEngineMethodEventJSON = function(const This: TSMObject; const Args: RawUTF8): RawUTF8 of object;
  // JSON-based callback signature used for TSMEngine.RegisterMethod()
  // - any Delphi exception raised during this execution will be converted into a JavaScript exception by
  //   TSMEngine
  // - similar to TServiceMethod.InternalExecute() as defined in mORMot.pas (for instance, this callback
  //   will be used to execute native Delphi interface-based methods from JavaScript code in
  //   mORMotSM.pas unit)
  // - “this” JavaScript calling object is transmitted as low-level TSMObject
  // - will expect as input a JSON array of parameters from Args, e.g.
  //   
  //   ```
  //   `[1,2,3]`
  //   ```
  //   - if the method only expect one result, shall return one JSON value, e.g.
  //     ```
  //     "6"
  //     ```
  //   - if the method expect more than one result (i.e. several var/out parameters in addition to the main
  //     function result), it shall return a JSON object, with parameter names for all var/out/result values, e.g.
  //     ```
  //     {
  //     "first":1,"second":2,"result":3
  //     }
  //     ```
- this allows the function result to be consumed by the JavaScript as a regular JS value or object
- corresponds to meJSON kind of callback method

TSMEngineMethodEventKind = ( meVariant, meJSON );

Kinds of callback methods available for TSMEngine.RegisterMethod()

TSMEngineMethodEventVariant = function( const This: variant; const Args: array of variant): variant of object;

/ variant-based callback signature used for TSMEngine.RegisterMethod() /
- any Delphi exception raised during this execution will be converted into a JavaScript exception by TSMEngine
- "this" JavaScript calling object is transmitted as a TSMVariant custom variant: you can use late-binding over it to access its methods or properties, or transtype it using TSMVariantData(Instance) and access its low-level API content
- input arguments (and function result) are simple variant values, or TDocVariant custom variant instance for any object as complex document
- corresponds to meVariant kind of callback method

TSMValues = array[0..(MaxInt div sizeof(TSMValue)) - 1] of TSMValue;

A jsval wrappers array

Constants implemented in the SynSM unit

STACK_CHUNK_SIZE: cardinal = 8192;

Default stack growing size, in bytes

Functions or procedures implemented in the SynSM unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSError</td>
<td>To be used to catch Delphi exceptions inside JSNative function implementation</td>
<td>1571</td>
</tr>
<tr>
<td>VariantToJSVal</td>
<td>Convert a variant to a JavaScript value</td>
<td>1571</td>
</tr>
</tbody>
</table>

procedure JSError(cx: PJSContext; aException: Exception; const aContext: RawByteString='');

To be used to catch Delphi exceptions inside JSNative function implementation
- usage example:
  try
    doSomething();
    Result := JS_TRUE;
  except
    on E: Exception do begin
      JS_SET_RVAL(cx, vp, JSVAL_VOID);
      JSError(cx, E);
      Result := JS_FALSE;
    end;

function VariantToJSVal(cx: PJSContext; const Value: Variant): jsval;

Convert a variant to a JavaScript value
Variables implemented in the SynSM unit

SMVariantType: TSynInvokeableVariantType = nil;

*The internal custom variant type used to register TSMVariant*

SynSMLog: TSynLogClass=TSynLog;

*Define the TSynLog class used for logging for all our SynSM related units*
* - you may override it with TSQLLog, if available from mORMot.pas*
* - since not all exceptions are handled specifically by this unit, you may better use a common TSynLog class for the whole application or module*
27.34. SynSMAPI.pas unit

Purpose: SpiderMonkey *.h header port to Delphi
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the SynSMAPI unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

Objects implemented in the SynSMAPI unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FrameDescription</td>
<td>Defines a frame of a stack trace</td>
<td>1579</td>
</tr>
<tr>
<td>FrameDescriptionEx</td>
<td>Defines an extended frame of a stack trace</td>
<td>1579</td>
</tr>
<tr>
<td>JSCompartment</td>
<td>JavaScript execution compartment</td>
<td>1575</td>
</tr>
<tr>
<td>JSConstDoubleSpec</td>
<td>JS object constant definition</td>
<td>1578</td>
</tr>
<tr>
<td>JSContext</td>
<td>JavaScript execution context</td>
<td>1574</td>
</tr>
<tr>
<td>JSErrorFormatString</td>
<td>Used by JSErrorCallback() callback</td>
<td>1578</td>
</tr>
<tr>
<td>JSErrorReport</td>
<td>Internal structure used to report JavaScript errors</td>
<td>1577</td>
</tr>
<tr>
<td>JSFunctionSpec</td>
<td>Defines a single function for an object</td>
<td>1578</td>
</tr>
<tr>
<td>JSNativeWrapper</td>
<td>Wrappers to replace JSNatives for JSFunctionSpecs</td>
<td>1578</td>
</tr>
<tr>
<td>JSObject</td>
<td>JSObject is the type of JavaScript objects in the JSAPI</td>
<td>1577</td>
</tr>
<tr>
<td>JSPROPERTYDESC</td>
<td>Points to a JavaScript object property description ^JSPROPERTYDESC; defines</td>
<td>1579</td>
</tr>
<tr>
<td></td>
<td>a JavaScript object property description</td>
<td></td>
</tr>
</tbody>
</table>
Objects | Description | Page
--- | --- | ---
JSPropertyDescArray | Stores JavaScript object properties description | 1579
JSPropertyOpWrapper | Wrappers to replace PropertyOp for JSPROPERTYSpecs | 1578
JSPROPERTYSpec | Defines custom JSPROPERTY | 1578
JSRuntime | JavaScript execution runtime | 1575
JSStrictPropertyOpWrapper | Wrappers to replace StrictPropertyOp for JSPROPERTYSpecs | 1578
JSString | JavaScript string instance | 1576
jsval_layout | Low-level definition of the jsval internals | 1578
jsval_payload | Low-level definition of the jsval internals | 1578
StackDescription | Defines a stack trace | 1579
StackDescriptionEx | Defines to an extended stack trace | 1579

**JSContext = object(TObject)**

*JavaScript execution context*
- this object does not store anything, but just provide some helper methods to access a PPJSContext value via JS_*Context*(..) API functions

**function** InitStandardClasses(global: PJSObject): boolean;
  *Wrapper to JS_InitStandardClasses(@self,global)*

**function** NewJSString(const Value: RawUTF8): PJSString; overload;
  *Create a new JavaScript string instance from a given UTF-8 text*

**function** NewJSString(const Value: SynUnicode): PJSString; overload;
  *Create a new JavaScript string instance from a given UTF-16 text*

**function** NewJSString(TextWide: PWideChar; TextLen: integer): PJSString; overload;
  *Create a new JavaScript string instance from a given UTF-16 text buffer*

**function** NewJSString(TextAnsi: PAnsiChar; TextLen, CodePage: integer): PJSString; overload;
  *Create a new JavaScript string instance from a given Ansi text buffer*
  - will use the specified Ansi code page for the conversion
  - if CodePage is 0, will use the CurrentAnsiCodePage value

**function** Runtime: PJSRuntime;
  *Wrapper to JS_GetRuntime(@self)*

**function** VersionToString: RawUTF8;
  *Wrapper to JS_GetVersion() and string conversion*

**procedure** Destroy;
  *Wrapper to JS_DestroyContext(@self)*
**property** Options: TJSOptions read GetOptions write SetOptions;

  Wrapper to JS_GetOptions()/JS_SetOptions()

- due to a XE3 bug, you should use the GetOptions/SetOptions methods instead of this property, under this compiler revision

**property** PrivateData: Pointer read GetPrivate write SetPrivate;

  Wrapper to JS_GetContextPrivate()/JS_SetContextPrivate()

**property** Version: JSVersion read GetVersion;

  Wrapper to JS_GetVersion()/JS_SetVersion()

---

**JSRuntime = object(TObject)**

*JavaScript execution runtime*

- this object does not store anything, but just provide some helper methods to access a PJSRuntime value via JS_*Runtime*(..) API functions

**procedure** Destroy;

  Wrapper to JS_DestroyRuntime(@self)

**procedure** Lock;

  Wrapper to JS_LockRuntime(@self)

**procedure** Unlock;

  Wrapper to JS_UnLockRuntime(@self)

**property** PrivateData: Pointer read GetPrivate write SetPrivate;

  Wrapper to JS_GetRuntimePrivate()/JS_SetRuntimePrivate()

---

**JSCompartment = object(TObject)**

*JavaScript execution compartment*

- this object does not store anything, but just provide some helper methods to access a PJSRuntime value via JS_*Runtime*(..) API functions

**function** EnterCompartment(cx: PJSContext; target: PJSObject): PJSCompartment;

  *Initialize and enter a JavaScript execution compartment*

**procedure** Destroy;

  *Leave a JavaScript execution compartment*
**JavaScript string instance**
- this object does not store anything, but just provide some helper methods to access a PPJSString value via JS_*String*(..) API functions
- use function JSContext.NewJSString() to create a new instance for a given execution context
- to understand string in SpiderMonkey good point is comments in vm\String.h in short this is C structure:

```c
struct Data {
    size_t                      lengthAndFlags;    /* JSString */
    union {
        const jschar           *chars;              /* JSLinearString */
        JSString               *left;               /* JSRope */
    } u1;
    union {
        void inlineStorage[NUM_INLINE_CHARS];     /* JS(Inline|Short)String */
        struct {
            union {
                JSLinearString *base;               /* JSDependentString */
                JSString        *right;              /* JSRope */
                size_t         capacity;            /* JSFlatString (extensible) */
                size_t         externalType;         /* JSExternalString */
            } u2;
            JSString        *parent;             /* JSRope (temporary) */
            void           *externalClosure;     /* JSExternalString */
            size_t         reserved;            /* may use for bug 615290 */
        } u3;
    } s;
} d;
```

but in API there is no need to use this structure, only pointer to it, and high-level access to the SpiderMonkey API via this JSString wrapper

**function** ToJSVal: jsval;
*Get a jsval corresponding to this string*

**function** ToString(cx: PJSCContext): string;
*Get the Delphi string text corresponding to this string, for a given runtime execution context*

**function** ToSynUnicode(cx: PJSCContext): SynUnicode;
*Get the UTF-16 text corresponding to this string, for a given runtime execution context*

**function** ToUTF8(cx: PJSCContext): RawUTF8; overload;
*Get the UTF-8 text corresponding to this string, for a given runtime execution context*

**function** ToWideString(cx: PJSCContext): WideString;
*Get the UTF-16 text corresponding to this string, for a given runtime execution context*

**procedure** ToJSONString(cx: PJSCContext; W: TTextWriter);
*Get the text encoded as a UTF-8 JSON string*

**procedure** ToUTF8(cx: PJSCContext; var result: RawUTF8); overload;
*Get the UTF-8 text corresponding to this string, for a given runtime execution context*
- slightly faster overloaded method (avoid string assignment)
procedure ToUTF8(cx: PJSContext; W: TTextWriter); overload;
   Add UTF-8 text corresponding to this string to writer, without escaping

procedure ToVariant(cx: PJSContext; var Value: Variant);
   Get the UTF-16 text corresponding to this string as a variant, for a given runtime execution context
   - will store a SynUnicode value into the variant instance

JSObject = object(TObject)
   JSObject is the type of JavaScript objects in the JSAPI
   - this object does not store anything, but just provide some helper methods to access a PJSObject value via low-level API functions

function ToJSValue: jsval;
   Get a jsval corresponding to this object

JSErrorReport = record
   Internal structure used to report JavaScript errors
   column: uintN;
      Zero-based column index in line
   errorNumber: uintN;
      The error number, e.g. see js.msg
   exnType: int16;
      One of the JSEXnType constants
   filename: PCChar;
      Source file name, URL, etc., or null
   flags: uintN;
      Error/warning, etc.
   linebuf: PCChar;
      Offending source line without final #13
   lineno: uintN;
      Source line number
   messageArgs: PPjschar;
      Arguments for the error message
   originPrincipals: PJSPrincipals;
      See 'originPrincipals' comment above
   tokenptr: PCChar;
      Pointer to error token in linebuf
   uclinebuf: Pjschar;
      Unicode (original) line buffer
ucmessage: Pjschar;
   The (default) error message
uctokenptr: Pjschar;
   Unicode (original) token pointer

JSErrorFormatString = record
   Used by JSErrorCallback() callback
   argCount: uint16;
      The number of arguments to expand in the formatted error message
   exnType: int16;
      One of the JSEXnType constants above
   format: PCChar;
      The error format string (UTF-8 if js_CStringsAreUTF8)

JSConstDoubleSpec = record
   JS object constant definition

JSStrictPropertyOpWrapper = record
   Wrappers to replace StrictPropertyOp for JSPropertySpecs
   - This will allow us to pass one JSJitInfo per function with the property spec, without additional
   field overhead.

JSPropertyOpWrapper = record
   Wrappers to replace PropertyOp for JSPropertySpecs
   - This will allow us to pass one JSJitInfo per function with the property spec, without additional
   field overhead.

JSPropertySpec = record
   Defines custom JSProperty
   - This will allow us to pass one JSJitInfo per function with the property spec, without additional
   field overhead.

JSNativeWrapper = record
   Wrappers to replace JSNatives for JSFunctionSpecs
   - This will allow us to pass one JSJitInfo per function with the property spec, without additional
   field overhead.

JSFunctionSpec = record
   Defines a single function for an object

jsval_payload = record
   Low-level definition of the jsval internals
   - do not use directly

jsval_layout = record
   Low-level definition of the jsval internals
   - do not use directly
FrameDescription = record
  Defines a frame of a stack trace

StackDescription = record
  Defines a stack trace

FrameDescriptionEx = record
  Defines an extended frame of a stack trace

StackDescriptionEx = record
  Defines an extended stack trace

JSPROPERTYDesc = record
  Points to a JavaScript object property description
  PJSPropertyDesc = ^JSPROPERTYDesc; defines a JavaScript object property description

  alias: jsva1;
    Contains the alias ID if dfAlias is included in description flags

  flags: JSPROPERTYDescFlags;
    The property behavior

  id: jsva1;
    The ID of this property

  spare_notused: uint8;
    This item is never used

  value: jsva1;
    The JavaScript value of this property

JSPROPERTYDescArray = record
  Stores JavaScript object properties description

Types implemented in the SynSMAPI unit

int16 = Smallint;
  16 bit signed integer type for C APIs

int32 = Integer;
  32 bit signed integer type for C APIs

int8 = ShortInt;
  8 bit signed integer type for C APIs

JSACCESSMode = ( JSACC_PROTO, JSACC_WATCH, JSACC_READ, JSACC_WRITE, JSACC_LIMIT );
  Js_CheCkAccess mode enumeration

JSArrayBufferViewType = ( jsabTYPE_INT8, jsabTYPE_UINT8, jsabTYPE_INT16, jsabTYPE_UINT16, jsabTYPE_INT32, jsabTYPE_UINT32, jsabTYPE_FLOAT32, jsabTYPE_FLOAT64, jsabTYPE_UINT8_CLAMPED, jsabTYPE_DATAVIEW, jsabTYPE_MAX );
  The available types of elements in a typed array or data view
  - obj must have passed a JS_IsArrayBufferView/JS_Is*Array test, or somehow be known that it would pass such a test: it is an ArrayBufferView or a wrapper of an ArrayBufferView, and the unwrapping
will succeed.
- jsabTYPE_UINT8_CLAMPED is a special type that is a uint8_t, but assignments are clamped to [0,255]: treat the raw data type as a uint8_t.
- jsabTYPE_DATAVIEW is the type returned for a DataView. Note that there is no single element type in this case

```pascal
JSBool = JSIntn;
```

*Boolean type for variables and parameter types for SMAPI*
- Use JS_FALSE and JS_TRUE constants for clarity of target type in assignments

```pascal
jsbytecode = uint8;
```

*Used to store a JavaScript bytecode item*

```pascal
jschar = Word;
```

*JMS is the type of JavaScript "characters", the 16 bits elements that make up JavaScript strings (maybe not truly valid UTF-16)*
- It is a 16-bit unsigned integer type
- As required by the ECMAScript standard, ECMA 262-3 §4.3.16, JavaScript strings are arbitrary sequences of 16-bit values
- A string may contain unmatched surrogates, which are not valid UTF-16
- It may also contain zeroes (0)
- so we did not define it as WideChar, but as abstract Word element

```pascal
JSClass.checkAccess = function (cx: PJSContext; var obj: PJSObject; var id: jsid; mode: JSAccessMode; vp: Pjsval): JSBool;
cdecl;
```

*JSClass.checkAccess type: check whether obj[id] may be accessed per mode, returning false on error/exception, true on success with obj[id]'s last-got value in *vp, and its attributes in *attrsp. As for JSPropertyOp above, id is either a string or an int jsval.*

```pascal
JSClassInternal = procedure; cdecl;
```

*Returns the external-string finalizer index for this string, or -1 if it is an "internal" (native to JS engine) string. TODO - in current compiled version there is no this functions. seems like a BUG!!!!!!! JS_GetExternalStringGCType function JS_IsExternalString(rt: PJSRuntime; str: PJSString): intN; cdecl; external SpiderMonkeyLib; For detailed comments on the function pointer types, see jspubtd.h*

```pascal
JSContextCallback = function (cx: PJSContext; contextOp: uintN): JSBool; cdecl;
```

*Callback prototype for a given runtime context*
- the possible values for contextOp when the runtime calls the callback are defined by JSContextOp

```pascal
JSConvertOp = ( JSCONTEXT_NEW, JSCONTEXT DESTROY );
```

*Flag used for JSContextCallback() argument*
- JSCONTEXT_NEW: JS_NewContext successfully created a new JSContext instance. The callback can initialize the instance as required. If the callback returns false, the instance will be destroyed and JS_NewContext returns null. In this case the callback is not called again.
- JSCONTEXT_DESTROY: One of JS_DestroyContext* methods is called. The callback may perform its own cleanup and must always return true.
- Any other value: For future compatibility the callback must do nothing and return true in this case.

```pascal
JSCDebugErrorHook = function (cx: PJSContext; message: PCChar; report: PJSErrorReport;
Closure: pointer): JSBool; cdecl;

Callback type to be called when an error is triggered during script debugging

JSDebuggerHandler = function(cx: PJSContext; script: PJSScript; pc: pjsbytecode; rval: pjsval; closure: pointer): JSTrapStatus; cdecl;

Callback type to be called when a script is debugging

JSDeletePropertyOp = function(cx: PJSContext; var obj: PJSObject; var id: jsid; succeeded: PJSBool): JSBool; cdecl;

JSClass method prototype to delete a property named by id in obj
- note the jsid type -- id may be a string (Unicode property identifier) or an int (element index)
- the *succeeded out parameter, on success, is the JSVAL_TRUE. *succeeded is JSVAL_FALSE if obj[id] can't be deleted (because it's permanent)

JSDestroyScriptHook = procedure(fop: PJSFreeOp; script: PJSScript; callerdata: Pointer); cdecl;

Callback type to be called when a JavaScript debugging hook is released

jsdouble = Double;

Jsdouble is the internal type of numbers, i.e. floating-point values
- jsdouble is obsolete since JavaScript 1.8.7+ - instead we must use C double so let's do it for future now
- in all cases see JSFloat64 - NSPR's floating point type is always 64 bits.

JSEnumerateOp = function(cx: PJSContext; var obj: PJSObject): JSBool; cdecl;

The old-style JSClass.enumerate op should define all lazy properties not yet reflected in obj.

JSErrorCallback = function(userRef: Pointer; const locale: PCChar; const errorNumber: uintN): PJSErrorFormatString; cdecl;

Callback prototype for returning an execution error

JSErrorReporter = procedure(cx: PJSContext; _message: PCChar; report: PJSErrorReport); cdecl;

Callback prototype for reporting error for a given runtime context

JSEXnType = ( JSEXN_NONE, JSEXN_ERR, JSEXN_INTERNALERR, JSEXN_EVALERR, JSEXN_RANGEERR, JSEXN_REFERENCEERR, JSEXN_SYNTAXERR, JSEXN_TYPEERR, JSEXN_URIERR, JSEXN_LIMIT );

Possible exception types
- These types are part of a JSErrorFormatString structure
- They define which error to throw in case of a runtime error
- JSEXN_NONE marks an unthrowable error

JSFinalizeOp = procedure(cx: PJSContext; obj: PJSObject); cdecl;

Finalize obj, which the garbage collector has determined to be unreachable from other live objects or from GC roots. Obviously, finalizers must never store a reference to obj.

JSFloat64 = Double;

Internal type of numbers, i.e. floating-point values for SMAPI
- NSPR's floating point type is always 64 bits.

JSGCCallback = function(cx: PJSContext; status: JSGCStatus): JSBool; cdecl;

Callback prototype for a given runtime context garbage collection

JSGCStatus = ( JSGC_BEGIN, JSGC_END );

Flag used for callback for a given runtime context garbage collection
JSHasInstanceOp = function(cx: PJSContext; obj: PJSObject; const v: Pjsval; var bp: JSBool): JSBool; cdecl;

Callback used to check whether v is an instance of obj or not
- Return false on error or exception, true on success with JS_TRUE in bp if v is an instance of obj, JS_FALSE in bp otherwise.

jsid = size_t;

Jsid is a generic identifier for any JavaScript object property of method
- A jsid is an identifier for a property or method of an object which is either a 31-bit signed integer, internal string or object. If XML is enabled, there is an additional singleton jsid value; see JS_DEFAULT_XML_NAMESPACE_ID below. Finally, there is an additional jsid value, JSID_VOID, which does not occur in JS scripts but may be used to indicate the absence of a valid jsid.

JSInt16 = SmallInt;

16 bit signed integer type for SMAPI

JSInt32 = Integer;

32 bit signed integer type for SMAPI

JSInt64 = Int64;

64 bit signed integer type for SMAPI

JSInt8 = ShortInt;

8 bit signed integer type for SMAPI

JSInterruptHook = function(cx: PJSContext; script: PJSScript; pc: pjsbytecode; rval: pjsval; closure: pointer): JSTrapStatus; cdecl;

Callback type to be called when script debugging is interrupted

JSIntn = PtrInt;

Type appropriate for most signed integer variables for SMAPI
- They are guaranteed to be at least 16 bits, though various architectures may define them to be wider (e.g., 32 or even 64 bits). These types are never valid for fields of a structure.

JSIterateOp = ( JSENUMERATE_INIT, JSENUMERATE_INIT_ALL, JSENUMERATE_NEXT, JSENUMERATE_DESTROY );

This enum type is used to control the behavior of a JSObject property iterator function that has type JSNewEnumerate.
- JSENUMERATE_INIT A new, opaque iterator state should be allocated and stored in *statep. (You can use PRIVATE_TO_JSVAL() to tag the pointer to be stored). The number of properties that will be enumerated should be returned as an integer jsval in *idp, if idp is non-null, and provided the number of enumerable properties is known. If idp is non-null and the number of enumerable properties can't be computed in advance, *idp should be set to JSVAL_ZERO.
- JSENUMERATE_INIT_ALL Used identically to JSENUMERATE_INIT, but exposes all properties of the object regardless of enumerability.
- JSENUMERATE_NEXT A previously allocated opaque iterator state is passed in via statep. Return the next jsid in the iteration using *idp. The opaque iterator state pointed at by statep is destroyed and *statep is set to JSVAL_NULL if there are no properties left to enumerate.
- JSENUMERATE_DESTROY Destroy the opaque iterator state previously allocated in *statep by a call to this function when enum_op was JSENUMERATE_INIT or JSENUMERATE_INIT_ALL.

JSNative = function(cx: PJSContext; argc: uintN; vp: Pjsval): JSBool; cdecl;

Here we miss trace and debug-only function defenition callback typedef for native functions called by the JS VM
cx is the execution context
- argc is the number of supplied arguments
- vp[0] is the callee - see JS_CALLEE()
- vp[1] is the object instance - see JS_THIS()
- vp[2]..vp[argc+1] are the supplied arguments - see JS_ARGV_PTR()

JSNewEnumerateOp = function(cx: PJSContext; var obj: PJSObject; enum_op: JSIterateOp; statep: pjsval; idp: pjsid): JSBool; cdecl;

Function prototype used for callbacks that enumerate the properties of a JSObject
- The behavior depends on the value of enum_op
- The return value is used to indicate success, with a value of JS_FALSE indicating failure.

JSNewResolveOp = function(cx: PJSContext; var obj: PJSObject; var id: jsid; flags: uintN; var objp: PPJSObject): JSBool;
cdecl;

Function prototype used to resolve a lazy property named by id in obj by defining it directly in obj
- Like JSResolveOp, but flags provide contextual information as follows:
  - JSRESOLVE_QUALIFIED a qualified property id: obj.id or obj[id], not id
  - JSRESOLVE_ASSIGNING obj[id] is on the left-hand side of an assignment
  - JSRESOLVE_DETECTING ‘if (o.p)...’ or similar detection opcode sequence
  - JSRESOLVE_DECLARING var, const, or function prolog declaration opcode
  - JSRESOLVE_CLASSNAME class name used when constructing

- The *objp out parameter, on success, should be null to indicate that id was not resolved; and non-null, referring to obj or one of its prototypes, if id was resolved.
- This hook instead of JSResolveOp is called via the JSClass.resolve member if JSCCLASS_NEW_RESOLVE is set in JSClass.flags.
- Setting JSCCLASS_NEW_RESOLVE and JSCCLASS_NEW_RESOLVE_GETS_START further extends this hook by passing in the starting object on the prototype chain via *objp. Thus a resolve hook implementation may define the property id being resolved in the object in which the id was first sought, rather than in a prototype object whose class led to the resolve hook being called.
- When using JSCCLASS_NEW_RESOLVE_GETS_START, the resolve hook must therefore null *objp to signify "not resolved". With only JSCCLASS_NEW_RESOLVE and no JSCCLASS_NEW_RESOLVE_GETS_START, the hook can assume *objp is null on entry. This is not good practice, but enough existing hook implementations count on it that we can't break compatibility by passing the starting object in *objp without a new JSClass flag.

JSNewScriptHook = procedure(cx: PJSContext; filename: PCChar; lineno: uintn; script:PJSScript; fun: PJSFunction; callerdata: pointer); cdecl;

Callback type to be called when debugging a JavaScript context

JSONWriteCallback = function(const buf: Pjschar; len: uint32; data: pointer): JSBool; cdecl;

Used by JS_Stringify() method to incremently write the JSON content

JSOperationCallback = function(cx: PJSContext): JSBool; cdecl;

Generic operation callback prototype for a given runtime context

JSPackedBool = JSUint8;

Packed boolean type for variables and parameter types for SMAPI
- use JSPackedBool within structs where bitfields are not desireable but minimum and consistent overhead matters.

JSPROPERTYDESCFLAG = ( dfEnumerate, dfReadOnly, dfPermanent, dfAlias, dfException, dfError );

Define object property behavior
- JSPD_ENUMERATE means that the property is visible to for/in loop
- JSPD_READONLY means that the property assignment will trigger an error
- JSPD_PERMANENT means that the property cannot be deleted
- JSPD_ALIAS means that the property has an alias id
- JSPD_EXCEPTION means that an exception occurred fetching the property: in this case, value is an exception
- JSPD_ERROR means that the native getter returned JS_FALSE without throwing an exception

JSPROPERTYDESCFLAGS = set of JSPROPERTYDESCFLAG;

Define object property behaviors

JSPROPERTYOP = function(cx: PJSContext; var obj: PJSObject; var id: jsid; vp: pjsval): JSBool; cdecl;

JSCLASS (and js::ObjectOps where appropriate) function pointer typedefs JSClass method prototype to add, or get a property named by id in obj
- note the jsid id type -- id may be a string (Unicode property identifier) or an int (element index)
- the vp out parameter, on success, is the new property value after an add or get. After a successful delete, *vp is JSVAL_FALSE iff obj[id] can't be deleted (because it's permanent)

JSPTRDIFF = ptrdiff_t;

Type for pointer arithmetic difference for SMAPI
- Variables of this type are suitable for storing a pointer or pointer subtraction

JSRESOLVEOP = function(cx: PJSContext; var obj: PJSObject; var id: jsid): JSBool; cdecl;

Function prototype used to resolve a lazy property named by id in obj by defining it directly in obj
- Lazy properties are those reflected from some peer native property space (e.g., the DOM attributes for a given node reflected as obj) on demand.
- JS looks for a property in an object, and if not found, tries to resolve the given id. If resolve succeeds, the engine looks again in case resolve defined obj[id]. If no such property exists directly in obj, the process is repeated with obj's prototype, etc.
- NB: JSNewResolveOp provides a cheaper way to resolve lazy properties.

JSSIZE = size_t;

A type for representing the size of objects for SMAPI

JSSTRICTPROPERTYOP = function(cx: PJSContext; var obj: PJSObject; var id: jsid; strict: JSBool; vp: pjsval): cdecl;

JSClass method prototype to set a property named by id in obj, treating the assignment as strict mode code if strict is true
- note the jsid id type -- id may be a string (Unicode property identifier) or an int (element index)
- the vp out parameter, on success, is the new property value after the set

JSTRINGFINALIZEOP = procedure(cx: PJSContext; var obj: PJSString); cdecl;

Callback used by JS_AddExternalStringFinalizer and JS_RemoveExternalStringFinalizer to extend and reduce the set of string types finalized by the GC.

JSTHROWHOOK = function(cx: PJSContext; script: PJSScript; pc: pjsbytecode; rval: pjsval; closure: pointer): JSTRAPSTATUS; cdecl;

Callback type to be called when script throws an exception

JSTRACEOP = procedure(cx: PJSContext; argc: uintN; vp: Pjssval); cdecl;

Callback used for trace operation of a given class
- enumerate all traceable things reachable from obj's private data structure.
- For each such thing, a trace implementation must call one of the JS_Call*Tracer variants on the
thing.
- JSTraceOp implementation can assume that no other threads mutates object state. It must not change state of the object or corresponding native structures. The only exception for this rule is the case when the embedding needs a tight integration with GC. In that case the embedding can check if the traversal is a part of the marking phase through calling JS_IsGCMarkingTracer and apply a specific code like emptying caches or marking its native structures.

```pascal
JSTrapHandler = function(cx: PJSContext; script: PJSScript; pc: pjsbytecode; rval: pjsval; closure: jsval): JSTrapStatus;
    cdecl;
    Callback type to be called when a script is debugging and trapped

JSTrapStatus = ( JSTRAP_ERROR, JSTRAP_CONTINUE, JSTRAP_RETURN, JSTRAP_THROW, JSTRAP_LIMIT );
    JavaScript debugging trap status

JSType = ( JSTYPE_VOID, JSTYPE_OBJECT, JSTYPE_FUNCTION, JSTYPE_STRING, JSTYPE_NUMBER, JSTYPE_BOOLEAN, JSTYPE_NULL, JSTYPE_LIMIT );
    Result of typeof operator enumeration

JSTypeOfOp = function(cx: PJSContext; var obj: PJSObject): JSType;
    cdecl;
    Callback used to delegate typeof to an object so it can cloak a primitive or another object

JSUint16 = Word;
    16 bit unsigned integer type for SMAPI

JSUint32 = Cardinal;
    32 bit unsigned integer type for SMAPI

JSUint64 = QWord;
    64 bit unsigned integer type for SMAPI

JSUint8 = Byte;
    8 bit unsigned integer type for SMAPI

JSUintn = PtrUInt;
    Type appropriate for most unsigned integer variables for SMAPI
    - They are guaranteed to be at least 16 bits, though various architectures may define them to be wider (e.g., 32 or even 64 bits). These types are never valid for fields of a structure.

JSIntPtr = PtrUInt;
    Ordinal type used for pointer arithmetic
    - Variables of this type are suitable for storing a pointer or pointer subtraction

JSUptrdiff = JSIntPtr;
    Ordinal type used for pointer arithmetic
    - Variables of this type are suitable for storing a pointer or pointer subtraction

JSUseHelperThreads = ( JS_NO_HELPER_THREADS, JS_USE_HELPER_THREADS );
    See http://developer.mozilla.org/en/docs/Category:JSAPI_Reference some aliases defined in SM:
    define JS_NewRuntime JS_Init
    define JS_DestroyRuntime JS_Finish
    define JS_LockRuntime JS_Lock
    define JS_UnlockRuntime JS_Unlock
    defines how function JS_NewRuntime() instantiate its threading model

JSUword = PtrUInt;
```
A JSWord is an unsigned integer that is the same size as a pointer

```pascal
jsval = type QWord;
```

Map a generic JavaScript value internal representation
- a variable of type jsval can hold exactly the same values as a JavaScript var or property: string, number, object, boolean, null, or undefined (including Arrays, functions, and Errors are all objects)
- jsval is a variant type whose exact representation varies by architecture.
- you should never use this value internals, but pjsval and its TSMValue wrapper, as defined in SynSM.pas unit, for a given TSMEngine execution context - see https://developer.mozilla.org/en-US/docs/SpiderMonkey/JSAPI_Reference/JSval

```pascal
JSVersion = Integer;
  Used to store a JavaScript version

JSWord = PtrInt;
  A JSWord is a signed integer that is the same size as a pointer

JSXDRObjectOp = function(xdr: PJSXDRState; var objp: PJSObject): JSBool; cdecl;
  Encode or decode an object, given an XDR state record representing external data
```

```pascal
PCChar = PAnsiChar;
  Define SpiderMonkey dedicated text buffer type
  - in SM 1.8.5 exist mode JS_CSringAreUTF8, so it is possible to use our RawUTF8 strings (in all cases internaly in SM it will be converted to Unicode - so need to test if SynCommons conversion faster or not-
  - but in SM 1.8.8 JS_CSringAreUTF8 removed, and all API may have to switch to jschar ( = Word/WideChar )?

PFrameDescription = ^FrameDescription;
  Points to a frame description of a stack trace

PFrameDescriptionEx = ^FrameDescriptionEx;
  Points to an extended frame description of a stack trace

PJSAtom = Pointer;
  Stores a JavaScript Atom

PJSBool = ^JSBool;
  Pointer to boolean type for variables and parameter types for SMAPI

pjsbytecode = ^jsbytecode;
  Used to store a pointer to JavaScript bytecode items

Pjschar = ^jschar;
  Pointer to a sequence of JavaScript "characters", i.e. some 16 bits elements that make up JavaScript strings (may not be truly valid UTF-16)
  - As required by the ECMA2Script standard, ECMA 262-3 §4.3.16, JavaScript strings are arbitrary sequences of 16-bit values
  - A string may contain unmatched surrogates, which are not valid UTF-16
  - It may also contain zeroes (0)

PJSCompartment = ^JSCompartment;
  Points to a JavaScript execution compartment
  - allows convenient access of JSCompartment wrapper methods
PJSContext = ^JSContext;
  * Pointer to JavaScript execution context
  * allows convenient access of JSContext wrapper methods

PJSErrorFormatString = ^JSErrorFormatString;
  * Pointer used by JSErrorCallback() callback

PJSErrorReport = ^JSErrorReport;
  * Map an internal structure used to report JavaScript errors

PJSFreeOp = Pointer;
  * Callback type to be called when an operation is freed

PJSFunction = type Pointer;
  * Abstract pointer to a JavaScript function

pjsid = ^jsid;
  * Pointer to a JavaScript object property of method identifier

PJSIdArray = ^JSIdArray;
  * Actually, length jsid words */

PjsidVector = ^TjsidVector;
  * Map an array to JavaScript object property of method identifiers

PJSJitInfo = pointer;
  * JS object property definition

PJSObject = ^JSObject;
  * Pointer to a JS Object instance

PJSPrincipals = Pointer;
  * Security protocol

PJSRuntime = ^JSRuntime;
  * Pointer to a JS Runtime instance

PJSJSScript = type Pointer;
  * Pointer to JSScript structure defined if jsscript.h
  * - we do not directly use of this structure, so we define just a pointer

PJSString = ^JSString;
  * Pointer to a JS String instance

PJSStringFinalizer = ^JSStringFinalizer;
  * * Finalizes external strings created by JS_NewExternalString.

PJSTracer = Pointer;
  * A JavaScript tracer instance

PJSUintptr = ^JSUintptr;
  * Pointer on an ordinal type used for pointer arithmetic

pjsval = ^jsval;
  * Pointer to a jsval JavaScript value

PjsvalVector = ^TjsvalVector;
Map an array of jsval JavaScript values

```pascal
PJSXDRState = type Pointer;
  State value as expected by JSXDRObjecOp() prototype

PPjschar = ^Pjschar;
  Pointer to a pointer of JavaScript "characters"
  - is mostly used for arrays of JavaScript strings

PPJSContext = ^PJSContext;
  Pointer to a pointer of JavaScript execution context

PPJSObject = ^PJSObject;
  Pointer to a pointer of JavaScript object

PPJSString = ^PJSString;
  Pointer to a pointer of JavaScript string

PRCondVar = Pointer;
  A event resource as defined by NSPR

PRIntervalTime = PRUint32;
  Interval time type as defined by NSPR

PRLock = Pointer;
  A mutex/lock resource as defined by NSPR

PRStatus = ( PR_FAILURE, PR_SUCCESS );
  Status codes as defined by NSPR

PRThread = Pointer;
  A thread resource as defined by NSPR

PRThreadPriority = ( PR_PRIORITY_FIRST, PR_PRIORITY_LOW, PR_PRIORITY_NORMAL,
  PR_PRIORITY_HIGH, PR_PRIORITY_URGENT, PR_PRIORITY_LAST );
  Thread priority as defined by NSPR
  - PR_PRIORITY_LOW is the lowest possible priority
  - PR_PRIORITY_NORMAL is the most common expected priority
  - PR_PRIORITY_HIGH is the slightly more aggressive scheduling
  - PR_PRIORITY_URGENT is there because it does little good to have one more priority value

PRThreadScope = ( PR_LOCAL_THREAD, PR_GLOBAL_THREAD, PR_GLOBAL_BOUND_THREAD );
  Thread scope as defined by NSPR

PRThreadState = ( PR_JOINABLE_THREAD, PR_UNJOINABLE_THREAD );
  Thread state as defined by NSPR

PRThreadType = ( PR_USER_THREAD, PR_SYSTEM_THREAD );
  Thread type as defined by NSPR

PRUint32 = uint32;
  Unsigned 32 bit integer type as defined by NSPR

PStackDescription = ^StackDescription;
  Points to a stack trace

PStackDescriptionEx = ^StackDescriptionEx;
```
Points to an extended stack trace

ptrdiff_t = PtrInt;
Type for pointer arithmetic difference
- Variables of this type are suitable for storing a pointer or pointer subtraction

puintN = ^uintn;
Pointer to a flag set variable

size_t = PtrUInt;
Variable type used to store a buffer size (in bytes) for SMAPI

TjsidVector = array[0..(MaxInt div sizeof(jsid))]-2 of jsid;
Abstract array to JavaScript object property of method identifiers
- set to -2 instead of -1 to allow JSIdArray record compilation

TJSOption = ( jsoExtraWarning, jsoWError, jsoVarObjFix, jsoPrivateIsNSISupports, jsoCompileNGo, jsoUnused5, jsoUnused6, jsoDontReportUncaught, jsoUnused9, jsoUnused10, jsoUnused11, jsoNoScriptRVal, jsoUnrootedGlobal, jsoBaseline, jsoPcCount, jsoTypeInference, jsoStrictMode, jsoIon, jsoAsmJS );
Available options for JS Objects

TJSOptions = set of TJSOption;
Set of available options for JS Objects

TJSPropertyAttr = ( jspEnumerate, jspReadOnly, jspPermanent, jspUnused, jspGetter, jspSetter, jspShared, jspIndex, jspShortID );
Available options for JS Objects Properties

TJSPropertyAttrs = set of TJSPropertyAttr;
Set of available options for JS Objects Properties

TjsvalVector = array[0..(MaxInt div sizeof(jsval))-1] of jsval;
An abstract array of jsval JavaScript values

uint16 = Word;
16 bit unsigned integer type for C APIs

uint32 = Cardinal;
32 bit unsigned integer type for C APIs

uint8 = Byte;
8 bit unsigned integer type for C APIs

uintn = PtrUInt;
Type appropriate for most flag set variables
- They are guaranteed to be at least 16 bits, though various architectures may define them to be wider (e.g., 32 or even 64 bits). These types are never valid for fields of a structure.

Constants implemented in the SynSMAPI unit

JSCLASS_EMULATES_UNDEFINED = 1 shl 6;
JSClass instance objects of this class act like the value undefined, in some contexts

JSCLASS_HAS_PRIVATE = 1 shl 0;
JSClass instance objects have private slot
JSCLASS.IMPLEMENTS_BARRIERS = 1 shl 5;
Correctly implements GC read and write barriers

JSCLASS.IS_DOM = 1 shl 4;
JSClass instance objects are DOM

JSCLASS.NEW_ENUMERATE = 1 shl 1;
JSClass instance has JSNewEnumerateOp hook

JSCLASS.NEW_RESOLVE = 1 shl 2;
JSClass instance has JSNewResolveOp hook

JSCLASS.PRIVATE.IS_NSISUPPORTS = 1 shl 3;
JSClass instance private is (nsISupports

JSCLASS.RESERVED_SLOTS.SHIFT = 8;
JSClass instance room for 8 flags below

JSCLASS.RESERVED_SLOTS.WIDTH = 8;
JSClass instance and 16 above this field

JSCLASS.USERBIT1 = 1 shl 7;
Reserved for embeddings.

JSFUN.GENERIC_NATIVE = $800;
* Specify a generic native prototype methods, i.e., methods of a class * prototype that are exposed as static methods taking an extra leading * argument: the generic |this| parameter. * * If you set this flag in a JSFunctionSpec struct's flags initializer, then * that struct must live at least as long as the native static method object * created due to this flag by JS_DefineFunctions or JS_InitClass. Typically * JSFunctionSpec structs are allocated in static arrays.

JSID_TYPE.STRING = $0;
A jsid is an identifier for a property or method of an object which is either a 31-bit signed integer, interned string or object. If XML is enabled, there is an additional singleton jsid value; see JS_DEFAULT_XMLNAMESPACE_ID below. Finally, there is an additional jsid value, JSID_VOID, which does not occur in JS scripts but may be used to indicate the absence of a valid jsid.

A jsid is not implicitly convertible to or from a jsval; JS_ValueToId or JS_IdToValue must be used instead.

JSNEW.ENUMERATE = 01;

TODO make jsid macro conversion  jsapi.h line 308-462  
define JSVAL_LOCK(cx,v) (JSVAL.IS_GCTHING(v)) \  
? JS_LockGCThing(cx, JSVAL_TO_GCTHING(v))  
define JSVAL_UNLOCK(cx,v) (JSVAL.IS_GCTHING(v)) \  
? JS_UnlockGCThing(cx, JSVAL_TO_GCTHING(v)) \  
Property attributes, set in JSPROPERTYSpec and passed to API functions

JSPROTO.LIMIT = 38;
JSPROTO.LIMIT is length of #include "jsproto.tbl"

JSREPORT.ERROR = 0;
JSErrorReporter function (set by JS_SetErrorReporter; see jspubtd.h for the JSErrorReporter typedef). JSErrorReport flag values. These may be freely composed. JSErrorReport pseudo-flag for default case
| JSREPORT_EXCEPTION | 2;     | JSErrorReport exception was thrown |
| JSREPORT_STRICT   | 4;     | JSErrorReport error or warning due to strict option |
| JSREPORT_STRICT_MODE_ERROR | 8;     | JSErrorReport error or warning depending on strict mode |
|                    |        | - This condition is an error in strict mode code, a warning if JS_HAS_STRICT_OPTION(cx), and otherwise should not be reported at all. We check the strictness of the context's top frame's script; where that isn't appropriate, the caller should do the right checks itself instead of using this flag. |
| JSREPORT_WARNING  | 1;     | JSErrorReport reported via JS_ReportWarning |
| JSRESOLVE_ASSIGNING | $02;   | Used by JS_ResolveStub() callback: resolve on the left of assignment |
| JSRESOLVE_CLASSNAME | $10;   | Used by JS_ResolveStub() callback: class name used when constructing |
| JSRESOLVE_DECLARING | $08;   | Used by JS_ResolveStub() callback: var, const, or function prolog op |
| JSRESOLVE_DETECTING | $04;   | Used by JS_ResolveStub() callback: 'if (o.p)...' or '{o.p}?...:' |
| JSRESOLVE_QUALIFIED | $01;   | Used by JS_ResolveStub() callback: resolve a qualified property id |
| JSRESOLVE_WITH     | $20;   | Used by JS_ResolveStub() callback: resolve inside a with statement |
| JSVERSION_1_0      | 100;   | Run-time version enumeration corresponding to 1.0 |
| JSVERSION_1_1      | 110;   | Run-time version enumeration corresponding to 1.1 |
| JSVERSION_1_2      | 120;   | Run-time version enumeration corresponding to 1.2 |
| JSVERSION_1_3      | 130;   | Run-time version enumeration corresponding to 1.3 |
| JSVERSION_1_4      | 140;   | Run-time version enumeration corresponding to 1.4 |
| JSVERSION_1_5      | 150;   | Run-time version enumeration corresponding to 1.5 |
| JSVERSION_1_6      | 160;   | Run-time version enumeration corresponding to 1.6 |
| JSVERSION_1_7      | 170;   | Run-time version enumeration corresponding to 1.7 |
JSVERSION_1_8 = 180;
Run-time version enumeration corresponding to 1.8

JSVERSION_DEFAULT = 0;
Run-time version enumeration corresponding to default version

JSVERSION_ECMA_3 = 148;
Run-time version enumeration corresponding to ECMA standard 3, i.e. 1.4.8

JSVERSION_ECMA_5 = 185;
Run-time version enumeration corresponding to ECMA standard 5, i.e. 1.8.5

JSVERSION_LATEST = JSVERSION_ECMA_5;
Run-time version enumeration corresponding to the latest available
that is, ECMA standard 5, i.e. 1.8.5

JSVERSION_UNKNOWN = -1;
Run-time version enumeration corresponding to an identified version

JS_DONTPRETTYPRINT: uintN = $8000;
API extension: OR this into indent to avoid pretty-printing the decompiled source resulting from JS_DecompileFunction{,Body}.

JS_FALSE = JSBool(0);
Boolean FALSE value for variables and parameter types for SMAPI

JS_TRUE = JSBool(1);
Boolean TRUE value for variables and parameter types for SMAPI

PRMJ_USEC_PER_SEC = 1000000;
Numbers of micro secs per second

PR_INTERVAL_NO_TIMEOUT =$ffffffff;
Stipulate that the process should wait forever as defined by NSPR
- i.e. will never time out
- defined in the PRIntervalTime namespace

PR_INTERVAL_NO_WAIT =$0;
Stipulate that the process should wait no time as defined by NSPR
- i.e. will return immediately
- defined in the PRIntervalTime namespace

Functions or procedures implemented in the SynSMAPI unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DescribeStack</td>
<td>Retrieve the stack trace of a given execution context</td>
<td>1607</td>
</tr>
<tr>
<td>DescribeStackEx</td>
<td>Retrieve the extended stack trace of a given execution context</td>
<td>1607</td>
</tr>
<tr>
<td>FormatStackDump</td>
<td>Dump stack trace info with the specified format</td>
<td>1607</td>
</tr>
<tr>
<td>FreeStackDescription</td>
<td>Release the stack trace description of a given execution context</td>
<td>1607</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>FreeStackDescriptionEx</td>
<td>Release the extended stack trace description of a given execution context</td>
<td>1607</td>
</tr>
<tr>
<td>JSVAL_IS_INT</td>
<td>Check if jsval is a 32 bit integer</td>
<td>1607</td>
</tr>
<tr>
<td>JSVAL_IS_NULL</td>
<td></td>
<td>1607</td>
</tr>
<tr>
<td>JSVAL_IS_VOID</td>
<td>Check if jsval is VOID</td>
<td>1607</td>
</tr>
<tr>
<td>JSVAL_TO_INT</td>
<td>Convert a jsval into a 32 bit integer</td>
<td>1607</td>
</tr>
<tr>
<td>JS_AddObjectRoot</td>
<td>Add a JSObject variable to the garbage collector’s root set</td>
<td>1607</td>
</tr>
<tr>
<td>JS_AddStringRoot</td>
<td>Add a JSString variable to the garbage collector’s root set</td>
<td>1608</td>
</tr>
<tr>
<td>JS_AddValueRoot</td>
<td>The JS_Add*Root functions add a C/C++ variable to the garbage collector’s root set, the set of variables used as starting points each time the collector checks to see what memory is reachable</td>
<td>1608</td>
</tr>
<tr>
<td>JS_AlreadyHasOwnProperty</td>
<td>JS_AliasElement OBSOLETE</td>
<td>1608</td>
</tr>
<tr>
<td>JS_AlreadyHasOwnProperty</td>
<td>JS_AliasProperty is Obsolete since JSAPI 8 function JS_AliasProperty(cx: PJSContext; obj: PJSObject; name: PAnsiChar; alias: PAnsiChar): JSBool; cdecl; external SpiderMonkeyLib ; Determine whether a property is already physically present on a JSObject</td>
<td>1608</td>
</tr>
<tr>
<td>JS_AlreadyHasOwnPropertyById</td>
<td>Determine whether a property is already physically present on a JSObject</td>
<td>1608</td>
</tr>
<tr>
<td>JS_AlreadyHasOwnPropertyUCProperty</td>
<td>Determine whether a property is already physically present on a JSObject</td>
<td>1609</td>
</tr>
<tr>
<td>JS_ARGV</td>
<td>Points to the first argument of a JSNative callback</td>
<td>1609</td>
</tr>
<tr>
<td>JS_AtomKey</td>
<td>Retrieve the local name into a JavaScript atom</td>
<td>1609</td>
</tr>
<tr>
<td>JS_BeginRequest</td>
<td>Indicates to the JS engine that the calling thread is entering a region of code that may call into the JSAPI but does not block</td>
<td>1609</td>
</tr>
<tr>
<td>JS_BufferIsCompilableUnit</td>
<td>Validate a JavaScript statement</td>
<td>1609</td>
</tr>
<tr>
<td>JS_CALLEE</td>
<td>Extern JS_PUBLIC_API(JSBool) JS_GetClassObject(JSContext *cx, JSObject *obj, JSProtoKey, JSObject **objp);</td>
<td>1609</td>
</tr>
<tr>
<td>JS_CallFunction</td>
<td>JS_CallFunction calls a specified function, fun, on an object, obj</td>
<td>1609</td>
</tr>
<tr>
<td>JS_CallFunctionName</td>
<td>Call a method of an object by name.</td>
<td>1610</td>
</tr>
<tr>
<td>JS_CallFunctionValue</td>
<td>Calls a specified JS function</td>
<td>1610</td>
</tr>
<tr>
<td>JS_CheckAccess</td>
<td>Check whether a running script may access a given object property.</td>
<td>1610</td>
</tr>
<tr>
<td>JS_ClearInterrupt</td>
<td>Clear a callback to be called when script debugging is interrupted</td>
<td>1610</td>
</tr>
<tr>
<td>JS_ClearTrap</td>
<td>Clear a trap debugging handler callback for a given execution context</td>
<td>1610</td>
</tr>
<tr>
<td>Function or procedure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>JS_CloneFunctionObject</td>
<td>JS_CloneFunctionObject creates a new function object from funobj</td>
<td>1610</td>
</tr>
<tr>
<td>JS_CompareStrings</td>
<td>Compares two JS strings, str1 and str2</td>
<td>1610</td>
</tr>
<tr>
<td>JS_CompileFunction</td>
<td>JS_CompileFunction compiles a function from a text string, bytes, and optionally associates it with a JS object, obj</td>
<td>1611</td>
</tr>
<tr>
<td>JS_CompileScript</td>
<td>JS_CompileScript compiles a script source, for execution</td>
<td>1611</td>
</tr>
<tr>
<td>JS_CompileUCFunction</td>
<td>JS_CompileUCFunction() is the Unicode version of JS_CompileFunction() function</td>
<td>1611</td>
</tr>
<tr>
<td>JS_CompileUCScript</td>
<td>Unicode version to compiles a script</td>
<td>1611</td>
</tr>
<tr>
<td>JS_ComputeThis</td>
<td>Low-level API used by JS_THIS() macro</td>
<td>1611</td>
</tr>
<tr>
<td>JS_ContextIterator</td>
<td>Cycles through the JS contexts associated with a particular JSRuntime</td>
<td>1611</td>
</tr>
<tr>
<td>JS_ConvertStub</td>
<td>Default callback matching JSConvertOp prototype of JSClass</td>
<td>1612</td>
</tr>
<tr>
<td>JS_ConvertValue</td>
<td>Converts a series of JS values, passed in an argument array, to their corresponding JS types</td>
<td>1612</td>
</tr>
<tr>
<td>JS_DecompileFunction</td>
<td>Generates the complete source code of a function declaration from a compiled function</td>
<td>1613</td>
</tr>
<tr>
<td>JS_DecompileFunctionBody</td>
<td>Generate the source code representing the body of a function, minus the function keyword, name, parameters, and braces</td>
<td>1613</td>
</tr>
<tr>
<td>JS_DecompileScript</td>
<td>Decompile a JavaScript script</td>
<td>1613</td>
</tr>
<tr>
<td>JS_DecompileScriptObject</td>
<td>Decompiles a Script Object back into its JavaScript representation</td>
<td>1613</td>
</tr>
<tr>
<td>JS_DeepFreezeObject</td>
<td>Freeze obj, and all objects it refers to, recursively</td>
<td>1613</td>
</tr>
<tr>
<td>JS_DefineConstDoubles</td>
<td>JS_DefineConstDoubles creates one or more properties for a specified object, obj, where each property consists of a double value.</td>
<td>1613</td>
</tr>
<tr>
<td>JS_DefineElement</td>
<td>JS_DefineElement defines a numeric property for a specified object, obj</td>
<td>1613</td>
</tr>
<tr>
<td>JS_DefineFunction</td>
<td>Create a native function and assign it as a property to a specified JS object</td>
<td>1613</td>
</tr>
<tr>
<td>JS_DefineFunctionById</td>
<td>Create a native function and assign it as a property to a specified JS object</td>
<td>1613</td>
</tr>
<tr>
<td>JS_DefineFunctions</td>
<td>JS_DefineFunctions creates zero or more functions and makes them properties (methods) of a specified object, obj, as if by calling JS_DefineFunction repeatedly</td>
<td>1614</td>
</tr>
<tr>
<td>JS_DefineObject</td>
<td>Create an object that is a property of another object</td>
<td>1614</td>
</tr>
<tr>
<td>JS_DefineOwnProperty</td>
<td>JS_DefineOwnProperty implements the ECMAScript defined function Object.defineProperty</td>
<td>1614</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>JS_DefineProperties</td>
<td>JS_DefineProperties creates properties on a specified object, obj. Each property is defined as though by calling JS_DefinePropertyWithTinyId.</td>
<td>1614</td>
</tr>
<tr>
<td>JS_DefineProperty</td>
<td>JS_DefineProperty is the fundamental operation on which several more convenient, higher-level functions are based, including JS_DefineFunction, JS_DefineFunctions, JS_DefineProperties, JS_DefineConstDoubles, JS_DefineObject, and JS_InitClass.</td>
<td>1614</td>
</tr>
<tr>
<td>JS_DefinePropertyById</td>
<td>JS_DefinePropertyById is the same as JS_DefineProperty but takes a jsid for the property name.</td>
<td>1614</td>
</tr>
<tr>
<td>JS_DefineUCFunction</td>
<td>Unicode version to create a native function.</td>
<td>1614</td>
</tr>
<tr>
<td>JS_DefineUCProperty</td>
<td>Define unicode property.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_DeleteElement</td>
<td>Removes a specified element or numeric property from an object.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_DeleteElement2</td>
<td>Removes a specified element or numeric property from an object.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_DeleteProperty</td>
<td>Removes a specified property from an object.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_DeleteProperty2</td>
<td>Removes a specified property from an object and return the property value.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_DeletePropertyById</td>
<td>Removes a specified property from an object.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_DeletePropertyById2</td>
<td>Removes a specified property from an object and return the property value.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_DeletePropertyStub</td>
<td>Default callback matching JSDeletePropertyOp prototype of JClass.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_DestroyContext</td>
<td>Release a new JContext.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_DestroyContextMaybeGC</td>
<td>Release a new JContext with potential garbage collection.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_DestroyContextNoGC</td>
<td>Release a new JContext without performing garbage collection.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_DestroyRuntime</td>
<td>Release the JavaScript runtime.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_DoubleIsInt32</td>
<td>Check if the given double is in fact a 31 bit signed integer.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_DropExceptionState</td>
<td>Drop a specified exception state.</td>
<td>1615</td>
</tr>
<tr>
<td>JS_EndRequest</td>
<td>Indicates to the JS engine that the calling thread is leaving a region of code that may call into the JSAPI but does not block.</td>
<td>1616</td>
</tr>
<tr>
<td>JS_EnterCompartment</td>
<td>Declare entering a safe compartment of the specified object.</td>
<td>1616</td>
</tr>
<tr>
<td>JS_Enumerate</td>
<td>JS_ClearScope - OBSOLETE JS_Enumerate gets the ids of all own properties of the specified object, obj,</td>
<td>1616</td>
</tr>
<tr>
<td>JS_EnumerateResolvedStandardClasses</td>
<td>Enumerate any already-resolved standard class ids into ida, or into a new JSIdArray if ida is null.</td>
<td>1616</td>
</tr>
<tr>
<td>JS_EnumerateStub</td>
<td>Default callback matching JSEnumerateOp prototype of JClass.</td>
<td>1616</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>JS_ErrorFromException</td>
<td>Retrieve an error report from an exception object</td>
<td>1616</td>
</tr>
<tr>
<td>JS_EvaluateScript</td>
<td>JS_EvaluateScript compiles and executes a script in the specified scope (obj)</td>
<td>1616</td>
</tr>
<tr>
<td>JS_evaluateUCInStackFrame</td>
<td>Compile and execute a script in stack frame with raw identifier (obj)</td>
<td>1617</td>
</tr>
<tr>
<td>JS_EvaluateUCScript</td>
<td>Unicode version of JS_EvaluateScript</td>
<td>1617</td>
</tr>
<tr>
<td>JS_ExecuteScript</td>
<td>Execute a compiled script.</td>
<td>1617</td>
</tr>
<tr>
<td>JS_FlushCaches</td>
<td>Flush the code cache for the current thread</td>
<td>1617</td>
</tr>
<tr>
<td>JS_FreezeObject</td>
<td>Freezes an object; see ES5’s Object.freeze(obj) method</td>
<td>1617</td>
</tr>
<tr>
<td>JS_GC</td>
<td>Launch the GarbageCollection process of the given execution RunTime</td>
<td>1618</td>
</tr>
<tr>
<td>JS_GetArrayBufferByteLength</td>
<td>Return the available byte length of an array buffer</td>
<td>1618</td>
</tr>
<tr>
<td>JS_GetArrayBufferData</td>
<td>Return a pointer to an array buffer's data</td>
<td>1618</td>
</tr>
<tr>
<td>JS_GetArrayBufferViewBuffer</td>
<td>Return the ArrayBuffer underlying an ArrayBufferView</td>
<td>1618</td>
</tr>
<tr>
<td>JS_GetArrayBufferViewByteLength</td>
<td>More generic name for JS_GetTypedArrayByteLength to cover DataView as well</td>
<td>1618</td>
</tr>
<tr>
<td>JS_GetArrayBufferViewData</td>
<td>Return a pointer to the start of the data referenced by any typed array</td>
<td>1618</td>
</tr>
<tr>
<td>JS_GetArrayBufferViewType</td>
<td>Get the type of elements in a typed array, or jsabTYPE_DATAVIEW if a DataView</td>
<td>1618</td>
</tr>
<tr>
<td>JS_GetArrayLength</td>
<td>Retrieve the length of an object array</td>
<td>1618</td>
</tr>
<tr>
<td>JS_GetClass</td>
<td>Retrieve the JSClass of a given object</td>
<td>1618</td>
</tr>
<tr>
<td>JS_GetContextPrivate</td>
<td>Read access to a JSContext field for application-specific data</td>
<td>1618</td>
</tr>
<tr>
<td>JS_GetDebugMode</td>
<td>Check if the JavaScript debugging mode is set for a given context</td>
<td>1618</td>
</tr>
<tr>
<td>JS_GetElement</td>
<td>Find a specified numeric property of an object and return its current value</td>
<td>1619</td>
</tr>
<tr>
<td>JS_GetFloat32ArrayData</td>
<td>Return a pointer to the start of the data referenced by a typed 32 bit float (single) array</td>
<td>1619</td>
</tr>
<tr>
<td>JS_GetFloat64ArrayData</td>
<td>Return a pointer to the start of the data referenced by a typed 64 bit float (double) array</td>
<td>1619</td>
</tr>
<tr>
<td>JS_GetFunctionArgumentCount</td>
<td>Retrieves how many arguments expect a function</td>
<td>1619</td>
</tr>
<tr>
<td>JS_GetFunctionArity</td>
<td>Return the arity (params count) for a specified function</td>
<td>1619</td>
</tr>
<tr>
<td>JS_GetFunctionFlags</td>
<td>Return JSFUN_* flags for a specified function</td>
<td>1619</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>JS_GetFunctionId</td>
<td>Return the function's identifier as a JSString, or null if fun is unnamed.</td>
<td>1619</td>
</tr>
<tr>
<td>JS_GetFunctionLocalNameArray</td>
<td>Retrieve the local name array information of a given function</td>
<td>1619</td>
</tr>
<tr>
<td>JS_GetFunctionObject</td>
<td>Retrieves the object for a specified function.</td>
<td>1619</td>
</tr>
<tr>
<td>JS_GetFunctionScript</td>
<td>Cast a JavaScript function into a script instance</td>
<td>1619</td>
</tr>
<tr>
<td>JS_GetInt16ArrayData</td>
<td>Return a pointer to the start of the data referenced by a typed 16 bit signed integer array</td>
<td>1620</td>
</tr>
<tr>
<td>JS_GetInt32ArrayData</td>
<td>Return a pointer to the start of the data referenced by a typed 32 bit signed integer array</td>
<td>1620</td>
</tr>
<tr>
<td>JS_GetInt8ArrayData</td>
<td>Return a pointer to the start of the data referenced by a typed 8 bit signed integer array</td>
<td>1620</td>
</tr>
<tr>
<td>JS_GetInternedStringChars</td>
<td>See JS_InternedString for details about what InternedString is.</td>
<td>1620</td>
</tr>
<tr>
<td>JS.GetObjectAsArrayBuffer</td>
<td>Unwrap an object as its raw binary memory buffer</td>
<td>1620</td>
</tr>
<tr>
<td>JS.GetObjectAsArrayBufferView</td>
<td>Unwrap an object as its raw binary memory buffer</td>
<td>1620</td>
</tr>
<tr>
<td>JSGetObjectAsFloat32Array</td>
<td>Unwrap 32 bit float (single) typed array into direct memory buffer</td>
<td>1620</td>
</tr>
<tr>
<td>JSGetObjectAsFloat64Array</td>
<td>Unwrap 64 bit float (double) typed array into direct memory buffer</td>
<td>1620</td>
</tr>
<tr>
<td>JSGetObjectAsInt16Array</td>
<td>Unwrap 16 bit signed integer typed array into direct memory buffer</td>
<td>1620</td>
</tr>
<tr>
<td>JSGetObjectAsInt32Array</td>
<td>Unwrap 32 bit signed integer typed array into direct memory buffer</td>
<td>1621</td>
</tr>
<tr>
<td>JSGetObjectAsInt8Array</td>
<td>Unwrap 8 bit signed integer typed array into direct memory buffer</td>
<td>1621</td>
</tr>
<tr>
<td>JSGetObjectAsUint16Array</td>
<td>Unwrap 16 bit unsigned integer typed array into direct memory buffer</td>
<td>1621</td>
</tr>
<tr>
<td>JSGetObjectAsUint32Array</td>
<td>Unwrap 32 bit unsigned integer typed array into direct memory buffer</td>
<td>1621</td>
</tr>
<tr>
<td>JSGetObjectAsUint8Array</td>
<td>Unwrap 8 bit unsigned integer typed array into direct memory buffer</td>
<td>1621</td>
</tr>
<tr>
<td>JSGetObjectAsUint8ClampedArray</td>
<td>Unwrap 8 bit unsigned integer array into direct memory buffer</td>
<td>1621</td>
</tr>
<tr>
<td>JS_GetObjectId</td>
<td>Get a unique identifier for obj, good for the lifetime of obj (even if it is moved by a copying GC)</td>
<td>1621</td>
</tr>
<tr>
<td>JS_GetOperationCallback</td>
<td>Retrieve the callback function that is automatically called periodically while JavaScript code runs in the given execution context</td>
<td>1621</td>
</tr>
</tbody>
</table>
### Functions or Procedures

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>JS_GetPendingException</td>
<td>Get the current exception being thrown within a context</td>
<td>1621</td>
</tr>
<tr>
<td>JS_GetProperty</td>
<td>Finds a specified property by name and retrieves its value</td>
<td>1622</td>
</tr>
<tr>
<td>JS_GetPropertyAttributes</td>
<td>Determine the attributes (JSPROP_* flags) of a property on a given object</td>
<td>1622</td>
</tr>
<tr>
<td>JS_GetPropertyById</td>
<td>Finds a specified property by jsid and retrieves its value</td>
<td>1622</td>
</tr>
<tr>
<td>JS_GetPropertyByIdDefault</td>
<td>Finds a specified property by jsid and retrieves its value or a default value</td>
<td>1622</td>
</tr>
<tr>
<td>JS_GetPropertyDescArray</td>
<td>Retrieve the description of a given JavaScript object property</td>
<td>1622</td>
</tr>
<tr>
<td>JS_GetPropertyDescriptorById</td>
<td>Finds a specified property of an object and gets a detailed description of that property</td>
<td>1622</td>
</tr>
<tr>
<td>JS_GetReservedSlot</td>
<td>Read access an object's reserved slots</td>
<td>1622</td>
</tr>
<tr>
<td>JS_GetRuntime</td>
<td>Retrieves a pointer to the JSRuntime with which a specified JSContext, is associated</td>
<td>1622</td>
</tr>
<tr>
<td>JS_GetScriptBaselineNumber</td>
<td>Retrieve the base line number of a given script</td>
<td>1623</td>
</tr>
<tr>
<td>JS_GetScriptLineExtent</td>
<td>Retrieve the extent of a given script</td>
<td>1623</td>
</tr>
<tr>
<td>JS_GetStringCharsAndLength</td>
<td>Retrieve a pointer to the 16-bit values that make up a given string.</td>
<td>1623</td>
</tr>
<tr>
<td>JS_GetStringCharsZ</td>
<td>Is the same as JS_GetStringChars except that it always returns either a null-terminated string or NULL, indicating out-of-memory</td>
<td>1623</td>
</tr>
<tr>
<td>JS_GetStringLength</td>
<td>TODO - no description provided in MSDN extern JS_PUBLIC_API(JSBool) JS_StringEqualsAscii(JSContext *cx, JSString *str, const char *asciiBytes, JSBool *match);</td>
<td>1624</td>
</tr>
<tr>
<td>JS_GetTypedArrayByteLength</td>
<td>Return the byte length of a typed array</td>
<td>1624</td>
</tr>
<tr>
<td>JS_GetTypedArrayByteOffset</td>
<td>Return the byte offset from the start of an array buffer to the start of a typed array view</td>
<td>1624</td>
</tr>
<tr>
<td>JS_GetTypedArrayLength</td>
<td>Return the number of elements in a typed array</td>
<td>1625</td>
</tr>
<tr>
<td>JS_GetTypeName</td>
<td>Determines the JS data type name of a JS value</td>
<td>1625</td>
</tr>
<tr>
<td>JS_GetUCPropertyAttributes</td>
<td>Determine the attributes (JSPROP_* flags) of a property on a given object</td>
<td>1625</td>
</tr>
<tr>
<td>JS_GetUint16ArrayData</td>
<td>Return a pointer to the start of the data referenced by a typed 16 bit unsigned integer array</td>
<td>1625</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>JS_GetUint32ArrayData</td>
<td>Return a pointer to the start of the data referenced by a typed 32 bit unsigned integer array</td>
<td>1625</td>
</tr>
<tr>
<td>JS_GetUint8ArrayData</td>
<td>Return a pointer to the start of the data referenced by a typed 8 bit unsigned integer array</td>
<td>1625</td>
</tr>
<tr>
<td>JS_GetUint8ClampedArrayData</td>
<td>Return a pointer to the start of the data referenced by a typed 8 bit unsigned integer array</td>
<td>1625</td>
</tr>
<tr>
<td>JS_GetVersion</td>
<td>Retrieve the JavaScript version number used within a specified executable script context</td>
<td>1625</td>
</tr>
<tr>
<td>JS_HasElement</td>
<td>Check if an object array has an element at the supplied index</td>
<td>1625</td>
</tr>
<tr>
<td>JS_HasProperty</td>
<td>JS_HasProperty searches an object, obj, and its prototype chain, for a property with the specified name</td>
<td>1626</td>
</tr>
<tr>
<td>JS_HasPropertyById</td>
<td>JS_HasProperty searches an object, obj, and its prototype chain, for a property with the specified jsid</td>
<td>1626</td>
</tr>
<tr>
<td>JS_HasUCProperty</td>
<td>Same as JS_HasProperty but unicode</td>
<td>1626</td>
</tr>
<tr>
<td>JS_InitClass</td>
<td>Make a JSClass accessible to JavaScript code by creating its prototype, constructor, properties, and functions.</td>
<td>1626</td>
</tr>
<tr>
<td>JS_InitStandardClasses</td>
<td>Get an interned string from a given null-terminated C string</td>
<td>1626</td>
</tr>
<tr>
<td>JS_InternJSString</td>
<td>Get an interned string from a given text buffer — that is, a JSString that is protected from GC and automatically shared with other code that needs a JSString with the same value</td>
<td>1626</td>
</tr>
<tr>
<td>JS_InternUCStringN</td>
<td>Unicode version of JS_InternString (faster)</td>
<td>1627</td>
</tr>
<tr>
<td>JS_IsAboutToBeFinalized</td>
<td>JS_IsAboutToBeFinalized() checks if the given object is going to be finalized at the end of the current GC</td>
<td>1627</td>
</tr>
<tr>
<td>JS_IsArrayBufferObjec t</td>
<td>Check whether obj supports the JS_GetArrayBuffer* APIs</td>
<td>1627</td>
</tr>
<tr>
<td>JS_IsArrayBufferViewOb ject</td>
<td>Check whether obj supports JS_GetArrayBufferView* APIs</td>
<td>1627</td>
</tr>
<tr>
<td>JS_IsArrayObject</td>
<td>Check if the supplied object is an array</td>
<td>1627</td>
</tr>
<tr>
<td>JS_IsConstructing</td>
<td>JS_IsConstructing must be called from within a native given the native’s original cx and vp arguments</td>
<td>1627</td>
</tr>
<tr>
<td>JS_IsExceptionPending</td>
<td>Determine whether an exception is pending in the JS engine.</td>
<td>1627</td>
</tr>
<tr>
<td>JS_IsExtensible</td>
<td>Queries the [[Extensible]] property of the object.</td>
<td>1627</td>
</tr>
<tr>
<td>JS_IsFloat32Array</td>
<td>Test for specific 32 bit float (single) typed array types (ArrayBufferView subtypes)</td>
<td>1628</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td><strong>JS_IsFloat64Array</strong></td>
<td>Test for specific 64 bit float (double) typed array types (ArrayBufferView subtypes)</td>
<td>1628</td>
</tr>
<tr>
<td><strong>JS_IsInRequest</strong></td>
<td>OBSOLETE procedure JS_YieldRequest(cx: PJSTextContext); cdecl; external SpiderMonkeyLib; OBSOLETE type OBSOLETE jsrefcount = JSInt32; OBSOLETE function JS_SuspendRequest(cx: PJSTextContext); jsrefcount; cdecl; external SpiderMonkeyLib; OBSOLETE procedure JS_ResumeRequest(cx: PJSTextContext; saveDepth: jsrefcount); cdecl; external SpiderMonkeyLib; checked is is within a code region protected by JS_BeginRequest()</td>
<td>1628</td>
</tr>
<tr>
<td><strong>JS_IsInt16Array</strong></td>
<td>Test for specific 16 bit signed integer typed array types (ArrayBufferView subtypes)</td>
<td>1628</td>
</tr>
<tr>
<td><strong>JS_IsInt32Array</strong></td>
<td>Test for specific 32 bit signed integer typed array types (ArrayBufferView subtypes)</td>
<td>1628</td>
</tr>
<tr>
<td><strong>JS_IsInt8Array</strong></td>
<td>Test for specific 8 bit signed integer typed array types (ArrayBufferView subtypes)</td>
<td>1628</td>
</tr>
<tr>
<td><strong>JS_IsRunning</strong></td>
<td>Determines if a script or function is currently executing in a specified JSContext, cx</td>
<td>1628</td>
</tr>
<tr>
<td><strong>JS_IsTypedArrayObject</strong></td>
<td>Check whether obj supports JS_GetTypedArray* APIs</td>
<td>1628</td>
</tr>
<tr>
<td><strong>JS_IsUint16Array</strong></td>
<td>Test for specific 16 bit unsigned integer typed array types (ArrayBufferView subtypes)</td>
<td>1628</td>
</tr>
<tr>
<td><strong>JS_IsUint32Array</strong></td>
<td>Test for specific 32 bit unsigned integer typed array types (ArrayBufferView subtypes)</td>
<td>1628</td>
</tr>
<tr>
<td><strong>JS_IsUint8Array</strong></td>
<td>Test for specific 8 bit unsigned integer typed array types (ArrayBufferView subtypes)</td>
<td>1628</td>
</tr>
<tr>
<td><strong>JS_IsUint8ClampedArray</strong></td>
<td>Test for specific 8 bit unsigned integer typed array types (ArrayBufferView subtypes)</td>
<td>1628</td>
</tr>
<tr>
<td><strong>JS_LeaveCompartment</strong></td>
<td>Declare leaving the safe compartment of the specified object</td>
<td>1628</td>
</tr>
<tr>
<td><strong>JS_LineNumberToPC</strong></td>
<td>Retrieve the byte code item position corresponding to a script line number</td>
<td>1628</td>
</tr>
<tr>
<td><strong>JS_LocalNameToAtom</strong></td>
<td>Convert a local name into a JavaScript atom</td>
<td>1629</td>
</tr>
<tr>
<td><strong>JS_LookupElement</strong></td>
<td>Determine if a specified numeric property exists</td>
<td>1629</td>
</tr>
<tr>
<td><strong>JS_LookupProperty</strong></td>
<td>Determine if a specified property exists, according to its name</td>
<td>1629</td>
</tr>
<tr>
<td><strong>JS_LookupPropertyById</strong></td>
<td>Determine if a specified property exists, according to its jsid</td>
<td>1629</td>
</tr>
<tr>
<td><strong>JS_LookupUCProperty</strong></td>
<td>Determine if a specified property exists</td>
<td>1629</td>
</tr>
<tr>
<td><strong>JS_MaybeGC</strong></td>
<td>Check if it would be worth it to launch the GarbageCollection process of the given execution RunTime, in the given context</td>
<td>1629</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>JS_New</td>
<td>As of SpiderMonkey 1.8.8, JS_ConstructObject and JS_ConstructObjectWithArguments have been removed from the JSAPI. The preferred alternative is to save a copy of the constructor function for the class, then to call it using JS_New. function JS_ConstructObject(cx: PJSContext; clasp: PJSClass; proto: PJSObject; parent: PJSObject; cdec; external SpiderMonkeyLib; function JS_ConstructObjectWithArguments(cx: PJSContext; clasp: PJSClass; proto: PJSObject; parent: PJSObject; argv: uintN; argv: pjsval): PJSObject; cdec; external SpiderMonkeyLib; Create an object as though by using the new keyword and a JavaScript function</td>
<td>1629</td>
</tr>
<tr>
<td>JS_NewArrayBuffer</td>
<td>Create a new ArrayBuffer with the given byte length.</td>
<td>1629</td>
</tr>
<tr>
<td>JS_NewArrayObject</td>
<td>JS_NewArrayObject creates a new array object with the specified length.</td>
<td>1630</td>
</tr>
<tr>
<td>JS_NewContext</td>
<td>Create a new JSContext</td>
<td>1630</td>
</tr>
<tr>
<td>JS_NewDateObject</td>
<td>Create a new JavaScript date object</td>
<td>1630</td>
</tr>
<tr>
<td>JS_NewDateObjectMsec</td>
<td>Create a new JavaScript date object from the Unix millisecond elapsed since EPOC</td>
<td>1630</td>
</tr>
<tr>
<td>JS_NewExternalString</td>
<td>Creates a new JSString whose characters are stored in external memory, i.e., memory allocated by the application, not the JavaScript engine</td>
<td>1630</td>
</tr>
<tr>
<td>JS_NewFloat32Array</td>
<td>Create a new signed 32 bit float (single) typed array with nelements</td>
<td>1630</td>
</tr>
<tr>
<td>JS_NewFloat32ArrayFromArray</td>
<td>Create a new 32 bit float (single) typed array and copy in values from a given object</td>
<td>1630</td>
</tr>
<tr>
<td>JS_NewFloat32ArrayWithBuffer</td>
<td>Create a new 32 bit float (single) typed array using the given ArrayBuffer for storage</td>
<td>1631</td>
</tr>
<tr>
<td>JS_NewFloat64Array</td>
<td>Create a new signed 64 bit float (double) typed array with nelements</td>
<td>1631</td>
</tr>
<tr>
<td>JS_NewFloat64ArrayFromArray</td>
<td>Create a new 64 bit float (double) typed array and copy in values from a given object</td>
<td>1631</td>
</tr>
<tr>
<td>JS_NewFloat64ArrayWithBuffer</td>
<td>Create a new 64 bit float (double) typed array using the given ArrayBuffer for storage</td>
<td>1631</td>
</tr>
<tr>
<td>JS_NewFunction</td>
<td>JS_NewFunction creates a new JavaScript function implemented in Delphi</td>
<td>1631</td>
</tr>
<tr>
<td>JS_NewFunctionById</td>
<td>Create the function with the name given by the id</td>
<td>1631</td>
</tr>
<tr>
<td>JS_NewGlobalObject</td>
<td>JS_NewGlobalObject creates a new global object based on the specified class</td>
<td>1631</td>
</tr>
<tr>
<td>JS_NewInt16Array</td>
<td>Create a new signed 16 bit integer typed array with nelements</td>
<td>1632</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>JS_NewInt16ArrayFromArray</td>
<td>Create a new 16 bit signed integer typed array and copy in values from a given object</td>
<td>1632</td>
</tr>
<tr>
<td>JS_NewInt16ArrayWithBuffer</td>
<td>Create a new 16 bit signed integer typed array using the given ArrayBuffer for storage</td>
<td>1632</td>
</tr>
<tr>
<td>JS_NewInt32Array</td>
<td>Create a new signed 32 bit integer typed array with nelements elements</td>
<td>1632</td>
</tr>
<tr>
<td>JS_NewInt32ArrayFromArray</td>
<td>Create a new 32 bit signed integer typed array and copy in values from a given object</td>
<td>1632</td>
</tr>
<tr>
<td>JS_NewInt32ArrayWithBuffer</td>
<td>Create a new 32 bit signed integer typed array using the given ArrayBuffer for storage</td>
<td>1632</td>
</tr>
<tr>
<td>JS_NewInt8Array</td>
<td>Create a new signed 8 bit integer typed array with nelements elements</td>
<td>1632</td>
</tr>
<tr>
<td>JS_NewInt8ArrayFromArray</td>
<td>Create a new 8 bit signed integer typed array and copy in values from a given object</td>
<td>1632</td>
</tr>
<tr>
<td>JS_NewInt8ArrayWithBuffer</td>
<td>Create a new 8 bit signed integer typed array using the given ArrayBuffer for storage</td>
<td>1632</td>
</tr>
<tr>
<td>JS_NewObject</td>
<td>JS_NewObject creates a new object based on a specified class, prototype, and parent object</td>
<td>1633</td>
</tr>
<tr>
<td>JS_NewObjectWithGivenProto</td>
<td>Unlike JS_NewObject, JS_NewObjectWithGivenProto does not compute a default proto if proto's actual parameter value is null</td>
<td>1633</td>
</tr>
<tr>
<td>JS_NewPropertyIterator</td>
<td>Create an object to iterate over enumerable properties of obj, in arbitrary</td>
<td>1633</td>
</tr>
<tr>
<td>JS_NewRuntime</td>
<td>Initialize the JavaScript runtime</td>
<td>1633</td>
</tr>
<tr>
<td>JS_NewStringCopyN</td>
<td></td>
<td>1633</td>
</tr>
<tr>
<td>JS_NewStringCopyZ</td>
<td>Create a new JavaScript string based on a null-terminated C string</td>
<td>1633</td>
</tr>
<tr>
<td>JS_NewUCString</td>
<td>Creates and returns a new string, using the memory starting at buf and ending at buf + length as the character storage.</td>
<td>1634</td>
</tr>
<tr>
<td>JS_NewUCStringCopyN</td>
<td>Unicode version of JS_NewStringCopyN</td>
<td>1634</td>
</tr>
<tr>
<td>JS_NewUCStringCopyZ</td>
<td>Unicode version of JS_NewStringCopyZ</td>
<td>1634</td>
</tr>
<tr>
<td>JS_NewUint16Array</td>
<td>Create a new unsigned 16 bit integer typed array with nelements elements</td>
<td>1634</td>
</tr>
<tr>
<td>JS_NewUint16ArrayFromArray</td>
<td>Create a new 16 bit unsigned integer typed array and copy in values from a given object</td>
<td>1634</td>
</tr>
<tr>
<td>JS_NewUint16ArrayWithBuffer</td>
<td>Create a new 16 bit unsigned integer typed array using the given ArrayBuffer for storage</td>
<td>1634</td>
</tr>
<tr>
<td>JS_NewUint32Array</td>
<td>Create a new unsigned 32 bit integer typed array with nelements elements</td>
<td>1634</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>JS_NewUint32ArrayFromArray</td>
<td>Create a new 32 bit unsigned integer typed array and copy in values from a given object</td>
<td>1634</td>
</tr>
<tr>
<td>JS_NewUint32ArrayBuffer</td>
<td>Create a new 32 bit unsigned integer typed array using the given ArrayBuffer for storage</td>
<td>1635</td>
</tr>
<tr>
<td>JS_NewUint8Array</td>
<td>Create a new unsigned 8 bit integer (byte) typed array with n elements</td>
<td>1635</td>
</tr>
<tr>
<td>JS_NewUint8ArrayFromArray</td>
<td>Create a new 8 bit unsigned integer typed array and copy in values from a given object</td>
<td>1635</td>
</tr>
<tr>
<td>JS_NewUint8ArrayWithBuffer</td>
<td>Create a new 8 bit unsigned integer typed array using the given ArrayBuffer for storage</td>
<td>1635</td>
</tr>
<tr>
<td>JS_NewUint8ClampedArray</td>
<td>Create a new 8 bit integer typed array with n elements</td>
<td>1635</td>
</tr>
<tr>
<td>JS_NewUint8ClampedArrayFromArray</td>
<td>Create a new 8 bit unsigned integer typed array and copy in values from a given object</td>
<td>1635</td>
</tr>
<tr>
<td>JS_NewUint8ClampedArrayWithBuffer</td>
<td>Create a new 8 bit unsigned integer typed array using the given ArrayBuffer for storage</td>
<td>1635</td>
</tr>
<tr>
<td>JS_NextProperty</td>
<td>Return true on success with *idp containing the id of the next enumerable</td>
<td>1635</td>
</tr>
<tr>
<td>JS_NumberValue</td>
<td>JS_NewNumberValue Obsolete since JavaScript mozjs17 use JS_NumberValue instead <a href="https://developer.mozilla.org/en-US/docs/SpiderMonkey/JSAPI_Reference/JS_NewNumberValue">https://developer.mozilla.org/en-US/docs/SpiderMonkey/JSAPI_Reference/JS_NewNumberValue</a></td>
<td>1635</td>
</tr>
<tr>
<td>JS_ObjectIsCallable</td>
<td>Check is the supplied object is callable</td>
<td>1636</td>
</tr>
<tr>
<td>JS_ObjectIsDate</td>
<td>Infallible predicate to test whether obj is a JavaScript date object</td>
<td>1636</td>
</tr>
<tr>
<td>JS_ObjectIsFunction</td>
<td>Test whether a given object is a Function</td>
<td>1636</td>
</tr>
<tr>
<td>JS_ParseJSON</td>
<td>Parse a string using the JSON syntax described in ECMAScript 5 and return the corresponding value into vp</td>
<td>1636</td>
</tr>
<tr>
<td>JS_PCToLineNumber</td>
<td>Retrieve the script line number from a byte code item position</td>
<td>1636</td>
</tr>
<tr>
<td>JS_PropertyStub</td>
<td>Default callback matching JSPropertyOp prototype of JSClass</td>
<td>1636</td>
</tr>
<tr>
<td>JS_PutPropertyDescriptorArray</td>
<td>Define the description of a given JavaScript object property</td>
<td>1636</td>
</tr>
<tr>
<td>JS_ReleaseFunctionLocalNameArray</td>
<td>Release the local name array information of a given function</td>
<td>1636</td>
</tr>
<tr>
<td>JS_RemoveObjectRoot</td>
<td>Remove a JSObject variable from the garbage collector's root set</td>
<td>1636</td>
</tr>
<tr>
<td>JS_RemoveStringRoot</td>
<td>Remove a JSString variable from the garbage collector's root set</td>
<td>1636</td>
</tr>
<tr>
<td>JS_RemoveValueRoot</td>
<td>Remove a jsval variable from the garbage collector's root set</td>
<td>1636</td>
</tr>
<tr>
<td>Function/Procedure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>JS_ReportAllocationOverflow</td>
<td>Call JS_ReportAllocationOverflow if an operation fails because it tries to use more memory (or more of some other resource) than the application is designed to handle</td>
<td>1637</td>
</tr>
<tr>
<td>JS_ReportError</td>
<td>JS_ReportError is the simplest JSAPI function for reporting errors</td>
<td>1637</td>
</tr>
<tr>
<td>JS_ReportErrorNumber</td>
<td>Report an error with an application-defined error code.</td>
<td>1637</td>
</tr>
<tr>
<td>JS_ReportErrorNumberUnc</td>
<td>Report an error with an application-defined error code.</td>
<td>1637</td>
</tr>
<tr>
<td>JS_ReportOutOfMemory</td>
<td>Reports a memory allocation error</td>
<td>1637</td>
</tr>
<tr>
<td>JS_ReportWarning</td>
<td>Similar to JS_ReportError(), but report a warning instead of an error (JSREPORT_JS_WARNING(report.flags))</td>
<td>1638</td>
</tr>
<tr>
<td>JS_ResolveStandardClass</td>
<td>Resolve id, which must contain either a string or an int, to a standard class name in obj if possible, defining the class's constructor and/or prototype and storing true in resolved</td>
<td>1638</td>
</tr>
<tr>
<td>JS.ResolveStub</td>
<td>Default callback matching JSSolveOp prototype of JSClass</td>
<td>1638</td>
</tr>
<tr>
<td>JS_RestoreExceptionState</td>
<td>Restore a specified exception state</td>
<td>1638</td>
</tr>
<tr>
<td>JS_RVAL</td>
<td>Get the return value of a JSNative callback</td>
<td>1638</td>
</tr>
<tr>
<td>JS_SameValue</td>
<td>Determines if two jsvals are the same, as determined by the SameValue algorithm in ECMAScript 262, 5th edition</td>
<td>1638</td>
</tr>
<tr>
<td>JS_SaveExceptionState</td>
<td>Save the current exception state. This takes a snapshot of cx's current exception state without making any change to that state.</td>
<td>1638</td>
</tr>
<tr>
<td>JS_SetArrayLength</td>
<td>JS_SetArrayLength sets the length property of an object obj</td>
<td>1639</td>
</tr>
<tr>
<td>JS_SetContextCallback</td>
<td>Specifies a callback function that is automatically called whenever a JSContext is created or destroyed</td>
<td>1639</td>
</tr>
<tr>
<td>JS_SetContextPrivate</td>
<td>Write access to a JSContext field for application-specific data</td>
<td>1639</td>
</tr>
<tr>
<td>JS_SetDebugErrorHook</td>
<td>Set a callback to be called when an error is triggered during script debugging</td>
<td>1639</td>
</tr>
<tr>
<td>JS_SetDebuggerHandler</td>
<td>Set a debugging handler callback for a given hook and runtime</td>
<td>1639</td>
</tr>
<tr>
<td>JS_SetDebugMode</td>
<td>Enables the JavaScript debugging mode for a given context</td>
<td>1639</td>
</tr>
<tr>
<td>JS_SetDestroyScriptHookProc</td>
<td>Set the callback to be called when a JavaScript debugging hook is released</td>
<td>1639</td>
</tr>
<tr>
<td>JS_SetElement</td>
<td>Assign a value to a numeric property of an object</td>
<td>1639</td>
</tr>
<tr>
<td>JS_SetErrorReporter</td>
<td>Specify the error reporting mechanism for an application.</td>
<td>1640</td>
</tr>
<tr>
<td>JS_SetGCCallback</td>
<td>Defines a Garbage Collection call-back function</td>
<td>1640</td>
</tr>
<tr>
<td>JS_SetInterrupt</td>
<td>Set a callback to be called when script debugging is interrupted</td>
<td>1640</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>JS_SettNativeStackQuota</td>
<td>Set the size of the native stack that should not be exceeded</td>
<td>1640</td>
</tr>
<tr>
<td>JS_SetNewScriptHookProc</td>
<td>Set a JavaScript debugging hook callback</td>
<td>1640</td>
</tr>
<tr>
<td>JS_SetOperationCallback</td>
<td>These functions allow setting an operation callback that will be called from the thread the context is associated with some time after any thread triggered the callback using JS_TriggerOperationCallback(cx).</td>
<td>1641</td>
</tr>
<tr>
<td>JS_SettPendingException</td>
<td>Set the current exception being thrown within a context</td>
<td>1641</td>
</tr>
<tr>
<td>JS_SettProperty</td>
<td>The following functions behave like JS_GetProperty and JS_GetPropertyById except when operating on E4X XML objects extern JS_PUBLIC_API(JSBool) JS_GetMethodById(JSContext *cx, JSObject *obj, jsid id, JSObject **objp, jsval *vp); extern JS_PUBLIC_API(JSBool) JS_GetMethod(JSContext *cx, JSObject *obj, const char *name, JSObject **objp, jsval *vp); JS_SetProperty assigns the value vp to the property name of the object obj</td>
<td>1641</td>
</tr>
<tr>
<td>JS_SettPropertyAttributes</td>
<td>Set the attributes of a property on a given object</td>
<td>1642</td>
</tr>
<tr>
<td>JS_SettPropertyById</td>
<td>JS_SetProperty assigns the value vp to the property jsid of the object obj</td>
<td>1642</td>
</tr>
<tr>
<td>JS_SettReservedSlot</td>
<td>Write access an object's reserved slots</td>
<td>1642</td>
</tr>
<tr>
<td>JS_SettRuntimeDebugMode</td>
<td>Enables the JavaScript debugging mode for a given runtime</td>
<td>1642</td>
</tr>
<tr>
<td>JS_SettSingleStepMode</td>
<td>Set single step mode. In this mode script interrupts on each line</td>
<td>1642</td>
</tr>
<tr>
<td>JS_SettThrowHook</td>
<td>Set a callback to be called when script throws an exception</td>
<td>1642</td>
</tr>
<tr>
<td>JS_SettTrap</td>
<td>Set a trap debugging handler callback for a given execution context</td>
<td>1642</td>
</tr>
<tr>
<td>JS_SettUCPropertyAttributes</td>
<td>Set the attributes of a property on a given object</td>
<td>1642</td>
</tr>
<tr>
<td>JS_SET_RVAL</td>
<td>Set the return value of a JSNative callback</td>
<td>1642</td>
</tr>
<tr>
<td>JS_ShutDown</td>
<td>Free all resources used by the JS engine, not associated with specific runtimes</td>
<td>1642</td>
</tr>
<tr>
<td>JS_StrictlyEqual</td>
<td>Determine whether two JavaScript values are equal in the sense of the === operator</td>
<td>1642</td>
</tr>
<tr>
<td>JS_StrictPropertyStub</td>
<td>Default callback matching JSStrictPropertyOp prototype of JClass</td>
<td>1642</td>
</tr>
<tr>
<td>JS_Stringify</td>
<td>Converts a value to JSON, optionally replacing values if a replacer function is specified, or optionally including only the specified properties if a replacer array is specified</td>
<td>1642</td>
</tr>
<tr>
<td>JS_StringToVersion</td>
<td>Configure a JSContext to use a specific version of the JavaScript language</td>
<td>1643</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>JS_THIS</td>
<td>Return the this object of a JSNative callback</td>
<td>1643</td>
</tr>
<tr>
<td>JS_THIS_OBJECT</td>
<td>Return the this object of a JSNative callback</td>
<td>1643</td>
</tr>
<tr>
<td>JS_ThrowReportedError</td>
<td>Given a reported error's message and JSErrorReport struct pointer, throw the corresponding exception on cx</td>
<td>1643</td>
</tr>
<tr>
<td>JS_ThrowStopIteration</td>
<td>Throws a StopIteration exception on cx</td>
<td>1643</td>
</tr>
<tr>
<td>JS_TriggerOperationCallback</td>
<td>Triggers a callback set using JS_SetOperationCallback</td>
<td>1643</td>
</tr>
<tr>
<td>JS_TypeOfValue</td>
<td>Determines the JS data type of a JS value</td>
<td>1643</td>
</tr>
<tr>
<td>JS_ValueToBoolean</td>
<td>Convert a JavaScript value to a boolean</td>
<td>1643</td>
</tr>
<tr>
<td>JS_ValueToECMAInt32</td>
<td>Convert a JavaScript value to an integer type as specified by the ECMAScript standard</td>
<td>1643</td>
</tr>
<tr>
<td>JS_ValueToECMAUint32</td>
<td>Convert a JavaScript value to an integer type as specified by the ECMAScript standard</td>
<td>1643</td>
</tr>
<tr>
<td>JS_ValueToFunction</td>
<td>Convert a jsval to a JSFunction</td>
<td>1643</td>
</tr>
<tr>
<td>JS_ValueToInt32</td>
<td>JS_ValueToInt32 is obsolete</td>
<td>1643</td>
</tr>
<tr>
<td>JS_ValueToNumber</td>
<td>Convert any JavaScript value to a floating-point number of type jsdouble.</td>
<td>1643</td>
</tr>
<tr>
<td>JS_ValueToObject</td>
<td>JS_ValueToObject converts a specified JavaScript value, v, to an object</td>
<td>1644</td>
</tr>
<tr>
<td>JS_ValueToSource</td>
<td>Convert any JavaScript value to its source representation</td>
<td>1644</td>
</tr>
<tr>
<td>JS_ValueToString</td>
<td>JS_ValueToString converts a specified JavaScript value, v, to a string.</td>
<td>1644</td>
</tr>
<tr>
<td>JS_ValueToUint16</td>
<td>Convert a JavaScript value to a 16 bit integer</td>
<td>1644</td>
</tr>
<tr>
<td>JS_VersionToString</td>
<td>JS_SetVersion is not supported now Use CompartmentOptions in JS_NewGlobalObject retrieve the JavaScript version text used within a specified executable script context</td>
<td>1645</td>
</tr>
<tr>
<td>PR_CreateThread</td>
<td>Initializes a NSPR thread</td>
<td>1645</td>
</tr>
<tr>
<td>PR_DestroyCondVar</td>
<td>Free a previously allocated NSPR event</td>
<td>1645</td>
</tr>
<tr>
<td>PR_DestroyLock</td>
<td>Free a previously allocated NSPR lock</td>
<td>1645</td>
</tr>
<tr>
<td>PR_JoinThread</td>
<td>Join a NSPR thread</td>
<td>1645</td>
</tr>
<tr>
<td>PR_Lock</td>
<td>Enter a previously allocated NSPR mutex/lock</td>
<td>1645</td>
</tr>
<tr>
<td>PR_NewCondVar</td>
<td>Allocates a new NSPR event</td>
<td>1645</td>
</tr>
<tr>
<td>PR_NewLock</td>
<td>Allocates a new NSPR mutex/lock</td>
<td>1645</td>
</tr>
<tr>
<td>PR_NotifyAllCondVar</td>
<td>Notify all previously allocated NSPR event</td>
<td>1645</td>
</tr>
</tbody>
</table>
### Functions or procedures

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR_NotifyCondVar</td>
<td>Notify a previously allocated NSPR event</td>
<td>1645</td>
</tr>
<tr>
<td>PR_SetCurrentThreadNa me</td>
<td>Change the current NSPR thread name</td>
<td>1645</td>
</tr>
<tr>
<td>PR_TicksPerSecond</td>
<td>Returns the number of ticks per second as expected by NSPR</td>
<td>1645</td>
</tr>
<tr>
<td>PR_Unlock</td>
<td>Leave a previously allocated NSPR mutex/lock</td>
<td>1645</td>
</tr>
<tr>
<td>PR_WaitCondVar</td>
<td>Wait until a previously allocated NSPR event is notified</td>
<td>1645</td>
</tr>
</tbody>
</table>

- **function** DescribeStack(cx: PJSContext; maxFrames: uintn): PStackDescription; cdecl; external SpiderMonkeyLib;
  - *Retrieve the stack trace of a given execution context*

- **function** DescribeStackEx(cx: PJSContext; maxFrames: uintn): PStackDescriptionEx; cdecl; external SpiderMonkeyLib;
  - *Retrieve the extended stack trace of a given execution context*

- **function** FormatStackDump(cx: PJSContext; buf: PCChar; showArgs, showLocals, showThisProps: JSBool): PCChar; cdecl; external SpiderMonkeyLib;
  - *Dump stack trace info with the specified format*

- **procedure** FreeStackDescription(cx: PJSContext; desc: PStackDescription); cdecl; external SpiderMonkeyLib;
  - *Release the stack trace description of a given execution context*

- **procedure** FreeStackDescriptionEx(cx: PJSContext; desc: PStackDescriptionEx); cdecl; external SpiderMonkeyLib;
  - *Release the extended stack trace description of a given execution context*

- **function** JSVAL_IS_INT(const v: jsval): Boolean;
  - *Check if jsval is a 32 bit integer*

- **function** JSVAL_IS_NULL(const v: jsval): Boolean;
  - *Check if jsval is NULL*

- **function** JSVAL_TO_INT(const v: jsval): jsint;
  - *Convert a jsval into a 32 bit integer*  
    - there is no check that jsval is really a 32 bit integer: caller shall ensure this is the case (by using

- **function** JS_AddObjectRoot(cx: PJSContext; rp: PPJSObject): JSBool; cdecl; external SpiderMonkeyLib;
  - *Add a JSObject variable to the garbage collector’s root set*  
    - similar to JS_AddValueRoot(), but with a JSObject value
function JS_AddStringRoot(cx: PJSContext; rp: PPJSString): JSBool; cdecl; external SpiderMonkeyLib;

Add a JSString variable to the garbage collector's root set
- similar to JS_AddValueRoot(), but with a JSString value

function JS_AddValueRoot(cx: PJSContext; vp: Pjsval): JSBool; cdecl; external SpiderMonkeyLib;

The JS_Add*Root functions add a C/C++ variable to the garbage collector's root set, the set of variables used as starting points each time the collector checks to see what memory is reachable
- The garbage collector aggressively collects and recycles memory that it deems unreachable, so roots are often necessary to protect data from being prematurely collected.
- vp/spp/opp/rp is the address of a C/C++ variable (or field, or array element) of type JSString *, JLObject *, or jsval. This variable must already be initialized. (For example, it must not be an uninitialized local variable. That could cause sporadic crashes during garbage collection, which can be hard to debug.) The variable must remain in memory until the balancing call to JS_RemoveRoot. Note that this means that if the root is meant to live past the end of a function, the address of a local (stack-based) variable may not be used for rp. If JS_Add*Root succeeds, then as long as this variable points to a JavaScript value or pointer to GC-thing, that value/GC-thing is protected from garbage collection. If the variable points to an object, then any memory reachable from its properties is automatically protected from garbage collection, too.
- Do not pass a pointer to a JString string or object to any of these functions — rp must point to a variable, the location of the pointer itself, and not an object or string.

function JS_AlreadyHasOwnElement(cx: PJSContext; obj: PJSObject; index: jsint; var foundp: JSBool): JSBool; cdecl; external SpiderMonkeyLib;

JS_AliasElement OBSOLETE

function JS_AlreadyHasOwnProperty(cx: PJSContext; obj: PJSObject; const name: PCChar; var found: JSBool): JSBool; cdecl; external SpiderMonkeyLib;

JS_AliasProperty is Obsolete since JSAPI 8 function JS_AliasProperty(cx: PJSContext; obj: PJSObject; name: PAnsiChar; alias: PAnsiChar): JSBool; cdecl; external SpiderMonkeyLib ; Determine whether a property is already physically present on a JQObject
- For native objects—objects whose properties are stored in the default data structure provided by SpiderMonkey, these functions simply check that data structure to see if the specified field is present. They do not search anywhere else for the property. This means that:
  - The prototype chain of obj is not searched.
  - The object's JSClass.resolve hook is not called, so lazily defined properties are not found. (This is the only API that can directly detect that a lazily resolved property has not yet been resolved.)
  - Shared, permanent, delegated properties are not found. (Such properties are an implementation detail of SpiderMonkey. They are meant to be a transparent optimization; this is the only API that breaks the abstraction)

function JS_AlreadyHasOwnPropertyById(cx: PJSContext; obj: PJSObject; id: jsid; var found: JSBool): JSBool; cdecl; external SpiderMonkeyLib;

Determine whether a property is already physically present on a JQObject
- this function will locate the property not by name, but by its jsid
**function** JS_AlreadyHasOwnUCProperty(cx: PJSContext; obj: PJSObject; const name: Pjschar; nameLen: size_t; var foundp: JSBool): JSBool; cdecl; external SpiderMonkeyLib;

**Determine whether a property is already physically present on a JSObject**
- For native objects—objects whose properties are stored in the default data structure provided by SpiderMonkey, these functions simply check that data structure to see if the specified field is present. They do not search anywhere else for the property. This means that: The prototype chain of obj is not searched. The object’s JSClass.resolve hook is not called, so lazily defined properties are not found. (This is the only API that can directly detect that a lazily resolved property has not yet been resolved.) Shared, permanent, delegated properties are not found.
- (Such properties are an implementation detail of SpiderMonkey. They are meant to be a transparent optimization; this is the only API that breaks the abstraction)

**function** JS_ARGV(cx: PJSContext; vp: Pjsval): PjsvalVector;
**Points to the first argument of a JSNative callback**

**function** JS_AtomKey(atom: PJSAtom): PJSString; cdecl; external SpiderMonkeyLib;
**Retrieve the local name into a JavaScript atom**

**procedure** JS_BeginRequest(cx: PJSContext); cdecl; external SpiderMonkeyLib;
**Indicates to the JS engine that the calling thread is entering a region of code that may call into the JSAPI but does not block**

**function** JS_BufferIsCompilableUnit(cx: PJSContext; obj: PJSObject; bytes: PCChar; length: size_t): JSBool; cdecl; external SpiderMonkeyLib;
**Validate a JavaScript statement**
- Given a buffer, return JS_FALSE if the buffer might become a valid javascript statement with the addition of more lines
- Otherwise return JS_TRUE. The intent is to support interactive compilation - accumulate
- lines in a buffer until JS_BufferIsCompilableUnit is true, then pass it to the compiler.

**function** JS_CALLEE(cx: PJSContext; vp: Pjsval): jsval;
**Extern JS_PUBLIC_API(JSBool) JS_GetClassObject(JSContext *cx, JSObject *obj, JSProtoKey key, JSObject **objp);**

** extern JS_PUBLIC_API(JSQObject *) JS_GetScopeChain(JSContext *cx, JSObject *obj);**

** extern JS_PUBLIC_API(JSQObject *) JS_GetGlobalForObject(JSContext *cx, JSObject *obj);**

**extern JS_PUBLIC_API(JSQObject *) JS_GetGlobalForScopeChain(JSContext *cx);** \#ifdef JS_HAS_CTYPES \- currently not shure we can use it. return the callee function of a JSNative callback

**function** JS_CallFunction(cx: PJSContext; this: PJSObject; fun: PJSFunction; argc: uintN; argv: pjsval; var rval: jsval): JSBool; cdecl; external SpiderMonkeyLib;
**JS_CallFunction calls a specified function, fun, on an object, obj**
- In terms of function execution, the object is treated as this
- Warning: Calling JS_CallFunction is safe only if the fun argument could be passed to
JS_GetFunctionObject safely: that is, it is a function implemented by a JSNative or JSFastNative or
the result of a call to JS_CompileFunction, JS_CompileUCFunction,
JS_CompileFunctionForPrincipals, or JS_CompileUCFunctionForPrincipals.
- Passing any other JSFunction pointer can lead to a crash or worse
function JS_CallFunctionName(cx: PJSContext; this: PJSObject; const name: PCChar; argc: uintN; argv: Pjsval; var rval: jsval): JSBool; cdecl; external SpiderMonkeyLib;

- Call a method of an object by name.
- JS_CallFunctionName executes a function-valued property, name, belonging to a specified JS object, obj

function JS_CallFunctionValue(cx: PJSContext; this: PJSObject; fval: jsval; argc: uintn; argv: Pjsval; var rval: jsval): JSBool; cdecl; external SpiderMonkeyLib;

- Calls a specified JS function
- fval is function value

function JS_CheckAccess(cx: PJSContext; obj: PJSObject; id: jsid; mode: JSAccessMode; var vp: jsval; var attrsp: uintN): JSBool; cdecl; external SpiderMonkeyLib;

- Check whether a running script may access a given object property.
- The access check proceeds as follows.
- If obj has custom JSObjectOps, the access check is delegated to the JSObjectOps.checkAccess callback.
- Otherwise, if obj's class has a non-null JSClass.checkAccess callback, then it is called to perform the check.
- Otherwise, if a runtime-wide check-object-access callback has been installed by calling JS_SetCheckObjectAccessCallback, then that is called to perform the check.
- Otherwise, access is granted.
- On success, JS_CheckAccess returns JS_TRUE, *vp is set to the current value stored value, of the specified property, and *attrsp is set to the property's attributes.
- On error or exception, including if access is denied, JS_CheckAccess returns JS_FALSE, and the values left in *vp and *attrsp are undefined

function JS_ClearInterrupt(rt: PJSRuntime; hook: PJSInterruptHook; closurep: PPointer): JSBool; cdecl; external SpiderMonkeyLib;

- Clear a callback to be called when script debugging is interrupted

procedure JS_ClearTrap(cx: PJSContext; script: PJSScript; pc: pjsbytecode; handler: JSTrapHandler; closure: jsval); cdecl; external SpiderMonkeyLib;

- Clear a trap debugging handler callback for a given execution context

function JS_CloneFunctionObject(cx: PJSContext; funobj: PJSObject; parent: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

- JS_CloneFunctionObject creates a new function object from funobj
- The new object has the same code and argument list as funobj, but uses parent as its enclosing scope.
- This can be helpful if funobj is an extant function that you wish to use as if it were enclosed by a newly-created global object

function JS_CompareStrings(cx: PJSContext; str1, str2: PJSString; var res: int32): JSBool; cdecl; external SpiderMonkeyLib;

- Compares two JS strings, str1 and str2
- If the strings are identical in content and length, JS_CompareStrings stores 0 in *result.
- If str1 is less than str2, *result is less than 0. (If str1 is greater than str2, *result is greater than 0.
- On success the function returns JS_TRUE.
- On error, it returns JS_FALSE and the value in result is unchanged.
- This function imposes a total order on all JavaScript strings, the same order imposed by the JavaScript string comparison operators (<, <=, >, >=), as described in ECMA 262-3 § 11.8.5.
function JS_CompileFunction(cx: PJSContext; obj: PJSObject; name: PCChar; nargs: uintN; argnames: PCStringVector; bytes: PCChar; length: size_t; filename: PCChar; lineno: uintN): PJSFunction; cdecl; external SpiderMonkeyLib;

JS_CompileFunction compiles a function from a text string, bytes, and optionally associates it with a JS object, obj.
- name is the name to assign to the newly created function.
- nargs is the number of arguments the function takes, and argnames is a pointer to the first element of an array of names to assign each argument.
- The number of argument names should match the number of arguments specified in nargs.
- body is a string containing the source code of the function.
- length is the length of the source code in characters.
- filename is the name of the file (or URL) containing the function. This information is used in error messages if an error occurs during compilation.
- Similarly, lineno is used to report the line number where an error occurred during compilation.
- If both obj and name are non-null, the new function becomes a method of obj (a new property is defined on obj with the given name and the new Function object as its value).

function JS_CompileScript(cx: PJSContext; obj: PJSObject; bytes: PCChar; length: size_t; filename: PCChar; lineno: uintN): PJSScript; cdecl; external SpiderMonkeyLib;

JS_CompileScript compiles a script source, for execution
- The script is associated with a JS object.
- source is the string containing the text of the script.
- length indicates the size of the text version of the script in characters.
- filename is the name of the file (or URL) containing the script.
- This information is included in error messages if an error occurs during compilation.
- Similarly, lineno is used to report the line number of the script or file where an error occurred during compilation.
- If the script is not part of a larger document, lineno should be 1 (as the first line of a file is universally considered to be line 1, not line 0).
- On success, JS_CompileScript and JS_CompileUCScript return an object representing the newly compiled script.
- Otherwise, they report an error and return NULL.
- To compile a script from an external file source rather than passing the actual script as an argument, use JS_CompileFile instead of JS_CompileScript.

function JS_CompileUCFunction(cx: PJSContext; obj: PJSObject; name: PCChar; nargs: uintN; argnames: PCStringVector; chars: Pjschar; length: size_t; filename: PCChar; lineno: uintN): PJSFunction; cdecl; external SpiderMonkeyLib;

JS_CompileUCFunction() is the Unicode version of JS_CompileFunction() function

function JS_CompileUCScript(cx: PJSContext; obj: PJSObject; chars: Pjschar; length: size_t; filename: PCChar; lineno: uintN): PJSScript; cdecl; external SpiderMonkeyLib;

Unicode version to compiles a script

function JS_ComputeThis(cx: PJSContext; vp: Pjsval): jsval; cdecl; external SpiderMonkeyLib;

Low-level API used by JS_THIS() macro

function JS_ContextIterator(rt: PJSRuntime; iterp: PPJSContext): PJSContext; cdecl; external SpiderMonkeyLib;

Cycles through the JS contexts associated with a particular JSRuntime
### function JS_ConvertStub

```pascal
function JS_ConvertStub(cx: PJSContext; var obj: PJSObject; _type: JSType; vp: pjsval): JSBool; cdecl; external SpiderMonkeyLib;
```

**Default callback matching JSConvertOp prototype of JSClass**

### function JS_ConvertValue

```pascal
function JS_ConvertValue(cx: PJSContext; v: jsval; _type: JSType; var vp: jsval): JSBool; cdecl; external SpiderMonkeyLib;
```

Converts a series of JS values, passed in an argument array, to their corresponding JS types.

- Format is a string of the following characters (spaces are insignificant), specifying the tabulated type conversions:

<table>
<thead>
<tr>
<th>Character</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>JSBool</td>
</tr>
<tr>
<td>c</td>
<td>uint16/jschar</td>
</tr>
<tr>
<td>i</td>
<td>int32</td>
</tr>
<tr>
<td>u</td>
<td>uint32</td>
</tr>
<tr>
<td>j</td>
<td>int32 (coordinate)</td>
</tr>
<tr>
<td>d</td>
<td>jsdouble</td>
</tr>
<tr>
<td>I</td>
<td>Integr</td>
</tr>
<tr>
<td>S</td>
<td>JSString *</td>
</tr>
<tr>
<td>W</td>
<td>jschar *</td>
</tr>
<tr>
<td>o</td>
<td>JSObject *</td>
</tr>
<tr>
<td>f</td>
<td>JSFunction *</td>
</tr>
<tr>
<td>v</td>
<td>jsval</td>
</tr>
<tr>
<td>*</td>
<td>N/A</td>
</tr>
<tr>
<td>/</td>
<td>End of required arguments</td>
</tr>
</tbody>
</table>

The variable argument list after format must consist of &b, &c, &s, e.g., where those variables have the types given above. For the pointer types char *, JSString *, and JSObject *, the pointed-at memory returned belongs to the JS runtime, not to the calling native code. The runtime promises to keep this memory valid so long as argv refers to allocated stack space (so long as the native function is active).

- Fewer arguments than format specifies may be passed only if there is a / in format after the last required argument specifier and argc is at least the number of required arguments. More arguments than format specifies may be passed without error; it is up to the caller to deal with trailing unconverted arguments. NOTE: we did not include this function yet, since it uses C varargs.

- JS_ConvertArguments

Function `JS_ConvertArguments` converts a JavaScript value, `v`, to the specified type.

- On success, the converted value is stored in `*vp`. Typically users of this function set `vp` to point to `v`, so that if conversion is successful, `v` now contains the converted value.

- On error or exception, it returns `JS_FALSE`, and the value left in `*vp` is undefined.
function JS_DecompileFunction(cx: PJSContext; fun: PJSFunction; indent: uintN): PJSString; cdecl; external SpiderMonkeyLib;

Generates the complete source code of a function declaration from a compiled function

function JS_DecompileFunctionBody(cx: PJSContext; fun: PJSFunction; indent: uintN): PJSString; cdecl; external SpiderMonkeyLib;

Generate the source code representing the body of a function, minus the function keyword, name, parameters, and braces

function JS_DecompileScript(cx: PJSContext; script: PJSScript; name: PCChar; indent: uintn ): PJSString; cdecl; external SpiderMonkeyLib;

Decompile a JavaScript script

function JS_DecompileScriptObject(cx: PJSContext; scriptObj: PJSObject; name: PCChar; indent: uintN): PJSString; cdecl; external SpiderMonkeyLib;

Decompiles a Script Object back into its JavaScript representation

function JS_DeepFreezeObject(cx: PJSContext; obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;

Freeze obj, and all objects it refers to, recursively
- This will not recurse through non-extensible objects, on the assumption that those are already deep-frozen.

function JS_DefineConstDoubles(cx: PJSContext; obj: PJSObject; cds: PJSConstDoubleSpec): JSBool; cdecl; external SpiderMonkeyLib;

JS_DefineConstDoubles creates one or more properties for a specified object, obj, where each property consists of a double value.
- Each property is automatically assigned attributes as specified in the flags field of the JSConstDoubleSpec structure pointed to by cds.
- If flags is set to 0, the attributes for the property are automatically set to JSPROP_PERMANENT | JSPROP_READONLY

function JS_DefineElement(cx: PJSContext; obj: PJSObject; index: jsint; value: jsval; getter: JSPropertyOp; setter: JSPropertyOp; attrs: uintN): JSBool; cdecl; external SpiderMonkeyLib;

JS_DefineElement defines a numeric property for a specified object, obj
- Starting in SpiderMonkey 1.8.5, jsval can store a full 32-bit integer, so index is any 32-bit integer

function JS_DefineFunction(cx: PJSContext; obj: PJSObject; name: PCChar; call: JSNative; nargs: uintN; attrs: uintN): PJSFunction; cdecl; external SpiderMonkeyLib;

Create a native function and assign it as a property to a specified JS object

function JS_DefineFunctionById(cx: PJSContext; obj: PJSObject; id: jsid; call: JSNative; nargs: uintN; attrs: uintN): PJSFunction; cdecl; external SpiderMonkeyLib;

Create a native function and assign it as a property to a specified JS object
function JS_DefineFunctions(cx: PJSContext; obj: PJSObject; fs: PJSFunctionSpec): JSBool; cdecl; external SpiderMonkeyLib;

  JS_DefineFunctions creates zero or more functions and makes them properties (methods) of a specified object, obj, as if by calling JS_DefineFunction repeatedly
  - fs is a pointer to the first element of an array of JSFunctionSpec records
  - This array is usually defined as a static global, with each record initialized using JS_FS or JS_FN.
  - Each array element defines a single function: its name, the native Delphi implementation, the number of JavaScript arguments the function expects, and any function flags and property attributes.
  - The last element of the array must contain 0 values

function JS_DefineObject(cx: PJSContext; obj: PJSObject; name: PCChar; clasp: PJSClass; proto: PJSObject; attrs: uintN): PJSObject; cdecl; external SpiderMonkeyLib;

  Create an object that is a property of another object

function JS_DefineOwnProperty(cx: PJSContext; obj: PJSObject; id: jsid; descriptor: jsval; bp: PJSBool): JSBool; cdecl; external SpiderMonkeyLib;

  JS_DefineOwnProperty implements the ECMAScript defined function Object.defineProperty
  - So the same restrictions apply as for that function (e.g. it is not possible to change a non-configurable property).
  - descriptor is supposed to be a property descriptor, this means you need to create an object with properties such as value, writable, get or set.
  - See Object.defineProperty for a list of possible fields.
  - A getter or setter defined with this functions will be observable from JS code.

function JS_DefineProperties(cx: PJSContext; obj: PJSObject; ps: PJSPropertySpec): JSBool; cdecl; external SpiderMonkeyLib;

  JS_DefineProperties creates properties on a specified object, obj. Each property is defined as though by calling JS_DefinePropertyWithTinyId

function JS_DefineProperty(cx: PJSContext; obj: PJSObject; const name: PCChar; value: jsval; getter: JSPROPERTYOp; setter: JSSTRICTPROPERTYOp; attrs: uintN): JSBool; cdecl; external SpiderMonkeyLib;

  JS_DefineProperty is the fundamental operation on which several more convenient, higher-level functions are based, including JS_DefineFunction, JS_DefineFunctions, JS_DefineProperties, JS_DefineConstDoubles, JS_DefineObject, and JS_InitClass.
  - It differs from JS_SetProperty in that: it does not behave like ordinary property assignment in the JavaScript language; it allows the application to specify additional details (getter, setter, and attrs) governing the new property's behavior; it never calls a setter; it can call the JSClass.addProperty callback when JS_SetProperty would not, because it can replace an existing property.
  - The parameters specify the new property's name, initial stored value, getter, setter, and property attributes (attrs).

function JS_DefinePropertyById(cx: PJSContext; obj: PJSObject; id: jsid; value: jsval; getter: JSPROPERTYOp; setter: JSSTRICTPROPERTYOp; attrs: uintN): JSBool; cdecl; external SpiderMonkeyLib;

  JS_DefinePropertyById is the same as JS_DefineProperty but takes a jsid for the property name

function JS_DefineUCFunction(cx: PJSContext; obj: PJSObject; name: Pjschar; namelen: size_t; call: JSNative; nargs: uintN; attrs: uintN): PJSFunction; cdecl; external SpiderMonkeyLib;

  Unicode version to create a native function
**function** JS_DefineUCProperty(cx: PJSContext; obj: PJSObject; const name: Pjschar; namelen: size_t; const value: jsval; getter: JSPropertyOp; setter: JSStrictPropertyOp; attrs: uintN): JSBool; cdecl; external SpiderMonkeyLib;

Define unicode property
- see JS_DefineProperty for details

**function** JS_DeleteElement(cx: PJSContext; obj: PJSObject; index: jsint): JSBool; cdecl; external SpiderMonkeyLib;

Removes a specified element or numeric property from an object

**function** JS_DeleteElement2(cx: PJSContext; obj: PJSObject; index: jsint; var rval: jsval): JSBool; cdecl; external SpiderMonkeyLib;

Removes a specified element or numeric property from an object
- this function also returns the deleted element

**function** JS_DeleteProperty(cx: PJSContext; obj: PJSObject; const name: PCChar): JSBool; cdecl; external SpiderMonkeyLib;

Removes a specified property from an object

**function** JS_DeleteProperty2(cx: PJSContext; obj: PJSObject; const name: PCChar; var rval: jsval): JSBool; cdecl; external SpiderMonkeyLib;

Removes a specified property from an object and return the property value

**function** JS_DeletePropertyById(cx: PJSContext; obj: PJSObject; id: jsid): JSBool; cdecl; external SpiderMonkeyLib;

Removes a specified property from an object

**function** JS_DeletePropertyById2(cx: PJSContext; obj: PJSObject; id: jsid; var rval: jsval): JSBool; cdecl; external SpiderMonkeyLib;

Removes a specified property from an object and return the property value

**function** JS_DeletePropertyStub(cx: PJSContext; var obj: PJSObject; var id: jsid; succeeded: PJSBool): JSBool; cdecl; external SpiderMonkeyLib;

Default callback matching JSDeletePropertyOp prototype of JClass

**procedure** JS_DestroyContext(cx: PJSContext); cdecl; external SpiderMonkeyLib;

Release a new JSContext

**procedure** JS_DestroyContextMaybeGC(cx: PJSContext); cdecl; external SpiderMonkeyLib;

Release a new JSContext with potential garbage collection
- this function may or may not perform garbage collection: the engine makes an educated guess as to whether enough memory would be reclaimed to justify the work

**procedure** JS_DestroyContextNoGC(cx: PJSContext); cdecl; external SpiderMonkeyLib;

Release a new JSContext without performing garbage collection

**procedure** JS_DestroyRuntime(rt: PJSRuntime); cdecl; external SpiderMonkeyLib;

Release the JavaScript runtime

**function** JS_DoubleIsInt32(d: jsdouble; var i: jsint): JSBool; cdecl; external SpiderMonkeyLib;

Check if the given double is in fact a 31 bit signed integer

**procedure** JS_DropExceptionState(cx: PJSContext; state: PJSExceptionState); cdecl; external SpiderMonkeyLib;

Drop a specified exception state
```pascal
procedure JS_EndRequest(cx: PJSContext); cdecl; external SpiderMonkeyLib;

Indicates to the JS engine that the calling thread is leaving a region of code that may call into the JSAPI but does not block

function JS_EnterCompartment(cx: PJSContext; target: PJSObject): PJSCompartment; cdecl; external SpiderMonkeyLib;

Declare entering a safe compartment of the specified object
- NB: This API is infallible; a NULL return value does not indicate error

function JS_Enumerate(cx: PJSContext; obj: PJSObject): PJSSIdArray; cdecl; external SpiderMonkeyLib;

JS_ClearScope - OBSOLETE JS_Enumerate gets the ids of all own properties of the specified object, obj,
- that have the JSPROP_ENUMERATE attribute. This calls obj's JSClass.enumerate hook.
- On success, JS_Enumerate returns a pointer to the first element of an array of property IDs.
- The application must free this array using JS_DestroyIdArray. On error or exception, JS_Enumerate returns NULL.
- Warning: The property ids in the returned JSIdArray are subject to garbage collection.
- Therefore a program that loops over the property ids must either root them all, ensure that the properties are not deleted (in a multithreaded program this requires even greater care), or ensure that garbage collection does not occur

function JS_EnumerateResolvedStandardClasses(cx: PJSContext; obj: PJSObject; ida: PJSSIdArray): PJSSIdArray; cdecl; external SpiderMonkeyLib;

Enumerate any already-resolved standard class ids into ida, or into a new JSIdArray if ida is null
- Return the augmented array on success, null on failure with ida (if it was non-null on entry) destroyed

function JS_EnumerateStub(cx: PJSContext; var obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;

Default callback matching JSEnumerateOp prototype of JSClass

function JS_ErrorFromException(cx: PJSContext; v: jsval): PJSErrorReport; cdecl; external SpiderMonkeyLib;

Retrieve an error report from an exception object
- If the given value is an exception object that originated from an error, the exception will contain an error report struct, and this API will return the address of that struct. Otherwise, it returns NULL.
- The lifetime of the error report struct that might be returned is the same as the lifetime of the exception object

function JS_EvaluateScript(cx: PJSContext; obj: PJSObject; bytes: PCChar; length: uintN; filename: PCChar; lineno: uintN; var rval: jsval): JSBool; cdecl; external SpiderMonkeyLib;

JS_EvaluateScript compiles and executes a script in the specified scope (obj)
- Warning: this function perform conversion from Ansi to unicode (internaly SM is unicode), so for best performance convert your scripts from RawUTF8/Ansi to unicode using SynCommonsUtf8DecodeToRawUnicode / WinAnsiConvert.AnsiToRawUnicode and use JS_EvaluateUCScript()
- For details about scope see JS_ExecuteScript
- We recomend to use JSACTION_VAROBJFIX and some of quality tools like jslint/jshint - jshint is prefered, because it suppotr ECMA5 (let strict and so on), jslint - only ECMA3
- to avoid declare variables in global use var x = 10 instead of x = 10
```
function JS_evaluateUCInStackFrame(cx: PJSContext; raw: JSuintptr; chars: Pjschar; length: size_t; filename: PCChar; lineno: uintN; var rval: jsval): boolean;
cdecl; external SpiderMonkeyLib;

Compile and execute a script in stack frame with raw identifier (obj)

function JS_EvaluateUCScript(cx: PJSContext; obj: PJSObject; chars: pjschar; length: uintN; filename: PCChar; lineno: uintN; var rval: jsval): JSBool;
cdecl; external SpiderMonkeyLib;

Unicode version of JS_EvaluateScript
- For details about scope see JS_ExecuteScript
- remember length is "the length of src, in characters" not in bytes for pjschar = PRawUnicode use length = (length(RawUnicode) shr 1)

function JS_ExecuteScript(cx: PJSContext; obj: PJSObject; scriptObj: PJSObject; var rval: jsval): JSBool;
cdecl; external SpiderMonkeyLib;

Execute a compiled script.
- On success, *rval receives the value from the last executed expression statement processed in the script
- NB: JS_ExecuteScript and the JS_Evaluate*Script* quadruplets use the obj parameter as the initial scope chain header, the 'this' keyword value, and the variables object (ECMA parlance for where 'var' and 'function' bind names) of the execution context for script.
- Using obj as the variables object is problematic if obj's parent (which is the scope chain link; see JS_SetParent and JS_NewObject) is not null: in this case, variables created by 'var x = 0', e.g., go in obj, but variables created by assignment to an unbound id, 'x = 0', go in the last object on the scope chain linked by parent.
- ECMA calls that last scoping object the "global object", but note that many embeddings have several such objects. ECMA requires that "global code" be executed with the variables object equal to this global object. But these JS API entry points provide freedom to execute code against a "sub-global", i.e., a parented or scoped object, in which case the variables object will differ from the last object on the scope chain, resulting in confusing and non-ECMA explicit vs. implicit variable creation.
- Caveat embedders: unless you already depend on this buggy variables object binding behavior, you should call JS_SetOptions(cx, JSOPTION_VAROBJFIX) or JS_SetOptions(cx, JS_GetOptions(cx) | JSOPTION_VAROBJFIX) -- the latter if someone may have set other options on cx already -- for each context in the application, if you pass parented objects as the obj parameter, or may ever pass such objects in the future.
- Why a runtime option? The alternative is to add six or so new API entry points with signatures matching the following six, and that doesn't seem worth the code bloat cost. Such new entry points would probably have less obvious names, too, so would not tend to be used. The JS_SetOption call, OTOH, can be more easily hacked into existing code that does not depend on the bug; such code can continue to use the familiar JS_EvaluateScript, etc., entry points.

procedure JS_FlushCaches(cx: PJSContext); cdecl; external SpiderMonkeyLib;

Flush the code cache for the current thread
- The operation might be delayed if the cache cannot be flushed currently because native code is currently executing.

function JS_FreezeObject(cx: PJSContext; obj: PJSObject): JSBool;
cdecl; external SpiderMonkeyLib;

Freezes an object; see ESS's Object.freeze(obj) method
procedure JS_GC(rt: PJSRuntime); cdecl; external SpiderMonkeyLib;

  Launch the GarbageCollection process of the given execution RunTime

function JS_GetArrayBufferByteLength(obj: PJSObject): uint32; cdecl; external SpiderMonkeyLib;

  Return the available byte length of an array buffer
  - obj must have passed a JS_IsArrayBufferObject test, or somehow be known that it would pass
    such a test: it is an ArrayBuffer or a wrapper of an ArrayBuffer, and the unwrapping will succeed

function JS_GetArrayBufferData(obj: PJSObject): Puint8Vector; cdecl; external SpiderMonkeyLib;

  Return a pointer to an array buffer's data
  - The buffer is still owned by the array buffer object, and should not be modified on another
    thread. The returned pointer is stable across GCs
  - obj must have passed a JS_IsArrayBufferObject test, or somehow be known that it would pass
    such a test: it is an ArrayBuffer or a wrapper of an ArrayBuffer, and the unwrapping will succeed.

function JS_GetArrayBufferViewBuffer(obj: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

  Return the ArrayBuffer underlying an ArrayBufferView
  - If the buffer has been neutered, this will still return the neutered buffer.
  - obj must be an object that would return true for JS_IsArrayBufferViewObject()

function JS_GetArrayBufferViewByteLength(obj: PJSObject): uint32; cdecl; external SpiderMonkeyLib;

  More generic name for JS_GetTypedArrayByteLength to cover DataViews as well

function JS_GetArrayBufferViewData(obj: PJSObject): Pointer; cdecl; external SpiderMonkeyLib;

  Return a pointer to the start of the data referenced by any typed array
  - The data is still owned by the typed array, and should not be modified on another thread
  - obj must have passed a JS_Is*Array test, or somehow be known that it would pass such a test: it
    is a typed array or a wrapper of a typed array, and the unwrapping will succeed
  - Prefer the type-specific versions when possible

function JS_GetArrayBufferViewType(obj: PJSObject): JSArrayBufferViewType; cdecl; external SpiderMonkeyLib;

  Get the type of elements in a typed array, or jsabTYPE_DATAVIEW if a DataView

function JS_GetArrayLength(cx: PJSContext; obj: PJSObject; var length: jsuint): JSBool; cdecl; external SpiderMonkeyLib;

  Retrieve the length of an object array

function JS_GetClass(obj: PJSObject): PJSClass; cdecl; external SpiderMonkeyLib name 'JS_GetClass';

  Retrieve the JSClass of a given object
  - JS_GetClass() expects only one parameter in every case

function JS_GetContextPrivate(cx: PJSContext): Pointer; cdecl; external SpiderMonkeyLib;

  Read access to a JSContext field for application-specific data

function JS_GetDebugMode(cx: PJSContext): JSBool; cdecl; external SpiderMonkeyLib;

  Check if the JavaScript debugging mode is set for a given context
function JS_GetElement(cx: PJSContext; obj: PJSObject; index: jsint; var vp: jsval): JSBool; cdecl; external SpiderMonkeyLib;

Find a specified numeric property of an object and return its current value

function JS_GetFloat32ArrayData(obj: PJSObject): Pfloat32Vector; cdecl; external SpiderMonkeyLib;

Return a pointer to the start of the data referenced by a typed 32 bit float (single) array
- The data is still owned by the typed array, and should not be modified on another thread
- obj must have passed a JS_Is*Array test, or somehow be known that it would pass such a test: it is a typed array or a wrapper of a typed array, and the unwrapping will succeed

function JS_GetFloat64ArrayData(obj: PJSObject): Pfloat64Vector; cdecl; external SpiderMonkeyLib;

Return a pointer to the start of the data referenced by a typed 64 bit float (double) array
- The data is still owned by the typed array, and should not be modified on another thread
- obj must have passed a JS_Is*Array test, or somehow be known that it would pass such a test: it is a typed array or a wrapper of a typed array, and the unwrapping will succeed

function JS_GetFunctionArgumentCount(cx: PJSContext; fun: PJSFunction): uint; cdecl; external SpiderMonkeyLib;

Retrieves how many arguments expect a function

function JS_GetFunctionArity(fun: PJSFunction): uint16; cdecl; external SpiderMonkeyLib;

Return the arity (params count) for a specified function

function JS_GetFunctionFlags(fun: PJSFunction): uintN; cdecl; external SpiderMonkeyLib;

Return JSFUN_* flags for a specified function

function JS_GetFunctionId(fun: PJSFunction): PJSString; cdecl; external SpiderMonkeyLib;

Return the function's identifier as a JSString, or null if fun is unnamed.
- The returned string lives as long as fun, so you don't need to root a saved
- reference to it if fun is well-connected or rooted, and provided you bound
- the use of the saved reference by fun's lifetime.

function JS_GetFunctionLocalNameArray(cx: PJSContext; fun: PJSFunction; var markp: pointer): PJSSuintptr; cdecl; external SpiderMonkeyLib;

Retrieve the local name array information of a given function

function JS_GetFunctionObject(fun: PJSFunction): PJSObject; cdecl; external SpiderMonkeyLib;

Retrieves the object for a specified function.
- fun should be a native function or JSAPI-compiled function! (result of a call to
  JS_CompileFunction, JS_CompileUCFunction)

function JS_GetFunctionScript(cx: PJSContext; fun: PJSFunction): PJSScript; cdecl; external SpiderMonkeyLib;

Cast a JavaScript function into a script instance
function JS_GetInt16ArrayData(obj: PJSObject): Pint16Vector; cdecl; external SpiderMonkeyLib;

Return a pointer to the start of the data referenced by a typed 16 bit signed integer array
- The data is still owned by the typed array, and should not be modified on another thread
- obj must have passed a JS_Is*Array test, or somehow be known that it would pass such a test: it is a typed array or a wrapper of a typed array, and the unwrapping will succeed

function JS_GetInt32ArrayData(obj: PJSObject): Pint32Vector; cdecl; external SpiderMonkeyLib;

Return a pointer to the start of the data referenced by a typed 32 bit signed integer array
- The data is still owned by the typed array, and should not be modified on another thread
- obj must have passed a JS_Is*Array test, or somehow be known that it would pass such a test: it is a typed array or a wrapper of a typed array, and the unwrapping will succeed

function JS_GetInt8ArrayData(obj: PJSObject): Pint8Vector; cdecl; external SpiderMonkeyLib;

Return a pointer to the start of the data referenced by a typed 8 bit signed integer array
- The data is still owned by the typed array, and should not be modified on another thread
- obj must have passed a JS_Is*Array test, or somehow be known that it would pass such a test: it is a typed array or a wrapper of a typed array, and the unwrapping will succeed

function JS_GetInternedStringChars(str: PJSString): Pjschar; cdecl; external SpiderMonkeyLib;
See JS_InternedString for details about what InternedString is.

function JS_GetObjectAsArrayBuffer(obj: PJSObject; var length: uint32; var Data: Puint8Vector): PJSObject; cdecl; external SpiderMonkeyLib;

Unwrap an object as its raw binary memory buffer
- Return nil without throwing any exception if the object cannot be viewed as the correct typed array, or the typed array object on success, filling both out parameters

function JS_GetObjectAsArrayBufferView(obj: PJSObject; var length: uint32; var Data: Puint8Vector): PJSObject; cdecl; external SpiderMonkeyLib;

Unwrap an object as its raw binary memory buffer
- Return nil without throwing any exception if the object cannot be viewed as the correct typed array, or the typed array object on success, filling both out parameters

function JS_GetObjectAsFloat32Array(obj: PJSObject; var length: uint32; var Data: Pfloat32Vector): PJSObject; cdecl; external SpiderMonkeyLib;

Unwrap 32 bit float (single) typed array into direct memory buffer
- Return nil without throwing any exception if the object cannot be viewed as the correct typed array, or the typed array object on success, filling both out parameters

function JS_GetObjectAsFloat64Array(obj: PJSObject; var length: uint32; var Data: Pfloat64Vector): PJSObject; cdecl; external SpiderMonkeyLib;

Unwrap 64 bit float (double) typed array into direct memory buffer
- Return nil without throwing any exception if the object cannot be viewed as the correct typed array, or the typed array object on success, filling both out parameters

function JS_GetObjectAsInt16Array(obj: PJSObject; var length: uint32; var Data: Pint16Vector): PJSObject; cdecl; external SpiderMonkeyLib;

Unwrap 16 bit signed integer typed array into direct memory buffer
- Return nil without throwing any exception if the object cannot be viewed as the correct typed array, or the typed array object on success, filling both out parameters
function JS_GetObjectAsInt32Array(obj: PJSObject; var length: uint32; var Data: Pint32Vector): PJSObject; cdecl; external SpiderMonkeyLib;

Unwrap 32 bit signed integer typed array into direct memory buffer
- Return nil without throwing any exception if the object cannot be viewed as the correct typed array, or the typed array object on success, filling both out parameters

function JS_GetObjectAsInt8Array(obj: PJSObject; var length: uint32; var Data: Pint8Vector): PJSObject; cdecl; external SpiderMonkeyLib;

Unwrap 8 bit signed integer typed array into direct memory buffer
- Return nil without throwing any exception if the object cannot be viewed as the correct typed array, or the typed array object on success, filling both out parameters

function JS_GetObjectAsUint16Array(obj: PJSObject; var length: uint32; var Data: Puint16Vector): PJSObject; cdecl; external SpiderMonkeyLib;

Unwrap 16 bit unsigned integer typed array into direct memory buffer
- Return nil without throwing any exception if the object cannot be viewed as the correct typed array, or the typed array object on success, filling both out parameters

function JS_GetObjectAsUint32Array(obj: PJSObject; var length: uint32; var Data: Puint32Vector): PJSObject; cdecl; external SpiderMonkeyLib;

Unwrap 32 bit unsigned integer typed array into direct memory buffer
- Return nil without throwing any exception if the object cannot be viewed as the correct typed array, or the typed array object on success, filling both out parameters

function JS_GetObjectAsUint8ClampedArray(obj: PJSObject; var length: uint32; var Data: Puint8Vector): PJSObject; cdecl; external SpiderMonkeyLib;

Unwrap 8 bit unsigned integer typed array into direct memory buffer
- Return nil without throwing any exception if the object cannot be viewed as the correct typed array, or the typed array object on success, filling both out parameters

function JS_GetObjectId(cx: PJSContext; obj: PJSObject; idp: Pjsid): JSBool; cdecl; external SpiderMonkeyLib;

Get a unique identifier for obj, good for the lifetime of obj (even if it is moved by a copying GC)
- Return false on failure (likely out of memory), and true with idp containing the unique id on success

function JS_GetOperationCallback(cx: PJSContext): JSOperationCallback; cdecl; external SpiderMonkeyLib;

Retrieve the callback function that is automatically called periodically while JavaScript code runs in the given execution context

function JS_GetPendingException(cx: PJSContext; var vp: jsval): JSBool; cdecl; external SpiderMonkeyLib;

Get the current exception being thrown within a context
function JS_GetProperty(cx: PJSContext; obj: PJSObject; const name: PCChar; var vp: jsval): JSBool; cdecl; external SpiderMonkeyLib;

* Finds a specified property by name and retrieves its value
  - JS_GetProperty examines a specified JS object obj and its prototype chain for a property with the specified name
  - It behaves like the JavaScript expression obj[name].

function JS_GetPropertyAttributes(cx: PJSContext; obj: PJSObject; const name: PCChar; var attrsp: uintN; var foundp: JSBool): JSBool; cdecl; external SpiderMonkeyLib;

* Determine the attributes (JSPROP_* flags) of a property on a given object
  - If the object does not have a property by that name, *foundp will be JS_FALSE and the value of *attrsp is undefined.

function JS_GetPropertyById(cx: PJSContext; obj: PJSObject; id: jsid; var vp: jsval): JSBool; cdecl; external SpiderMonkeyLib;

* Finds a specified property by jsid and retrieves its value
  - JS_GetProperty examines a specified JS object obj and its prototype chain for a property with the specified jsid

function JS_GetPropertyByIdDefault(cx: PJSContext; obj: PJSObject; id: jsid; def: jsval; var vp: jsval): JSBool; cdecl; external SpiderMonkeyLib;

* Finds a specified property by jsid and retrieves its value or a default value

function JS_GetPropertyDescArray(cx: PJSContext; obj: PJSObject; var pda: JSPropertyDescArray): JSBool; cdecl; external SpiderMonkeyLib;

* Retrieve the description of a given JavaScript object property

function JS_GetPropertyDescriptorById(cx: PJSContext; obj: PJSObject; id: jsid; flags: uintN; objp: PPJSObject; desc: PJSPropertyDescriptor): JSBool; cdecl; external SpiderMonkeyLib;

* Finds a specified property of an object and gets a detailed description of that property

function JS_GetReservedSlot(cx: PJSContext; obj: PJSObject; index: uint32; vp: pjsval): JSBool; cdecl; external SpiderMonkeyLib;

* Read access an object's reserved slots
  - If a JSClass has JSCLASS_HAS_RESERVED_SLOTS(n) in its flags, with n > 0, or has a non-null JSClass.reserveSlots callback, then objects of that class have n reserved slots in which the application may store data. These fields are not directly exposed to scripts.
  - Reserved slots may contain any jsvalue, and the garbage collector will hold the value alive as long as the object itself is alive
  - Reserved slots may also contain private values to store pointer values (whose lowest bit is 0) or uint32_t, when non-JavaScript values must be stored; the garbage collector ignores such values when it sees them.
  - Note that private values must not be exposed directly to JavaScript. It's only legal to store and retrieve data from private values. They cannot be returned from functions, set as properties on objects, embedded in JavaScript arrays, and so on
  - New objects' reserved slots are initialized to undefined.
  - TODO - collision in MSDN no parameter vp: pjsval but in jsapi.h is!

function JS_GetRuntime(cx: PJSContext): PJSRuntime; cdecl; external SpiderMonkeyLib;

* Retrieves a pointer to the JSRuntime with which a specified JSContext, is associated
function JS_GetScriptBaseLineNumber(cx: PJSContext; script: PJSScript): uint; cdecl;
external SpiderMonkeyLib;

    Retrieve the base line number of a given script

function JS_GetScriptLineExtent(cx: PJSContext; script: PJSScript): uint; cdecl;
external SpiderMonkeyLib;

    Retrieve the extent of a given script

function JS_GetStringCharsAndLength(cx: PJSContext; str: PJSString; var len: size_t): pjschar; cdecl; external SpiderMonkeyLib;

    Retrieve a pointer to the 16-bit values that make up a given string.
    - The array is not necessarily null-terminated. To get the length of the string, use
      JS_GetStringLength.
    - The program must not modify the array. If it does, the behavior is undefined.
    - The content of a JS string is not guaranteed to be valid UTF-16. It may contain surrogate code
      units that aren't properly paired. It may also contain zeroes.
    - The array returned by this function remains valid as long as str is valid. (Eventually, str becomes
      unreachable, the garbage collector collects it, and the array is freed by the system.)

function JS_GetStringCharsZ(cx: PJSContext; str: PJSString): pjschar; cdecl; external SpiderMonkeyLib;

    Is the same as JS_GetStringChars except that it always returns either a null-terminated string or
    NULL, indicating out-of-memory
    - help provided at
function JS_GetStringLength(str: PJSString): size_t; cdecl; external SpiderMonkeyLib;

TODO - no description provided in MSDN extern JS_PUBLIC_API(JSBool)
JS_StringEqualsAscii(cx, JString *str, const char *asciiBytes, JSBool *match);

extern JS_PUBLIC_API(size_t) JS_PutEscapedString(cx, char *buffer, size_t size, JString *str, char quote);
extern JS_PUBLIC_API(JSBool) JS_FileEscapedString(fp, JString *str, char quote);

Extracting string characters and length.

- While getting the length of a string is infallible, getting the chars can fail. As indicated by the lack of a JSContext parameter, there are two special cases where getting the chars is infallible:
  - The first case is interned strings, i.e., strings from JS_InternString or JSID_TO_STRING(id), using JS_GetInternedStringChars//.
  - The second case is "flat" strings that have been explicitly prepared in a fallible context by JS_FlattenString. To catch errors, a separate opaque JSFlatString type is returned by JS_FlattenString and expected by JS_GetFlatStringChars. Note, though, that this is purely a syntactic distinction: the input and output of JS_FlattenString are the same actual GC-thing so only one needs to be rooted. If a JString is known to be flat, JS_ASSERT_STRING_IS_FLAT can be used to make a debug-checked cast. Example: // in a fallible context JSFlatString *fstr = JS_FlattenString(cx, str); if (!fstr) return JS_FALSE; JS_ASSERT(fstr ==
  JS_ASSERT_STRING_IS_FLAT(str));

  // in an infallible context, for the same 'str' const jschar *chars = JS_GetFlatStringChars(fstr)
  JS_ASSERT(chars);

The CharsZ APIs guarantee that the returned array has a null character at chars[length]. This can require additional copying so clients should prefer APIs without CharsZ if possible. The infallible functions also return null-terminated arrays. (There is no additional cost or non-Z alternative for the infallible functions, so 'Z' is left out of the identifier.) Reports the length, in 16-bit code units, of the string str.

- This is the same as the length property of the string.
- This is the same as the length of the array returned by JS_GetStringChars, in jschars (not bytes).
- Because some Unicode characters are represented using two 16-bit code units, the result is not necessarily the same as the number of Unicode characters in the string.

function JS_GetTypedArrayByteLength(obj: PJSObject): uint32; cdecl; external SpiderMonkeyLib;

Return the byte length of a typed array
- obj must have passed a JS_IsTypedArrayObject/JS_Is*Array test, or somehow be known that it would pass such a test: it is a typed array or a wrapper of a typed array, and the unwrapping will succeed

function JS_GetTypedArrayByteOffset(obj: PJSObject): uint32; cdecl; external SpiderMonkeyLib;

Return the byte offset from the start of an array buffer to the start of a typed array view
- obj must have passed a JS_IsTypedArrayObject/JS_Is*Array test, or somehow be known that it would pass such a test: it is a typed array or a wrapper of a typed array, and the unwrapping will succeed.
function JS_GetTypedArrayLength(obj: PJSObject): uint32; cdecl; external SpiderMonkeyLib;

Return the number of elements in a typed array
- obj must have passed a JS_IsTypedArrayObject/JS_Is*Array test, or somehow be known that it would pass such a test: it is a typed array or a wrapper of a typed array, and the unwrapping will succeed.

function JS_GetTypeName(cx: PJSContext; _type: JSType): PCChar; cdecl; external SpiderMonkeyLib;

Determines the JS data type name of a JS value
- If the object does not have a property by that name, *foundp will be - JS_FALSE and the value of *attrsp is undefined.

function JS_GetUint16ArrayData(obj: PJSObject): Puint16Vector; cdecl; external SpiderMonkeyLib;

Return a pointer to the start of the data referenced by a typed 16 bit unsigned integer array
- The data is still owned by the typed array, and should not be modified on another thread
- obj must have passed a JS_Is*Array test, or somehow be known that it would pass such a test: it is a typed array or a wrapper of a typed array, and the unwrapping will succeed

function JS_GetUint32ArrayData(obj: PJSObject): Puint32Vector; cdecl; external SpiderMonkeyLib;

Return a pointer to the start of the data referenced by a typed 32 bit unsigned integer array
- The data is still owned by the typed array, and should not be modified on another thread
- obj must have passed a JS_Is*Array test, or somehow be known that it would pass such a test: it is a typed array or a wrapper of a typed array, and the unwrapping will succeed

function JS_GetUint8ArrayData(obj: PJSObject): Puint8Vector; cdecl; external SpiderMonkeyLib;

Return a pointer to the start of the data referenced by a typed 8 bit unsigned integer array
- The data is still owned by the typed array, and should not be modified on another thread
- obj must have passed a JS_Is*Array test, or somehow be known that it would pass such a test: it is a typed array or a wrapper of a typed array, and the unwrapping will succeed

function JS_GetUint8ClampedArrayData(obj: PJSObject): Puint8Vector; cdecl; external SpiderMonkeyLib;

Return a pointer to the start of the data referenced by a typed 8 bit unsigned integer array
- The data is still owned by the typed array, and should not be modified on another thread
- obj must have passed a JS_Is*Array test, or somehow be known that it would pass such a test: it is a typed array or a wrapper of a typed array, and the unwrapping will succeed

function JS_GetVersion(cx: PJSContext): JSVersion; cdecl; external SpiderMonkeyLib;

Retrieve the JavaScript version number used within a specified executable script context

function JS_HasElement(cx: PJSContext; obj: PJSObject; index: jsint; var foundp: JSBool): JSBool; cdecl; external SpiderMonkeyLib;

Check if an object array has an element at the supplied index
function JS_HasProperty(cx: PJSContext; obj: PJSObject; const name: PCChar; var found: JSBool): JSBool; cdecl; external SpiderMonkeyLib;

*JS_HasProperty searches an object, obj, and its prototype chain, for a property with the specified name*

- It behaves like the JavaScript expression name in obj

function JS_HasPropertyById(cx: PJSContext; obj: PJSObject; id: jsid; var found: JSBool): JSBool; cdecl; external SpiderMonkeyLib;

*JS_HasProperty searches an object, obj, and its prototype chain, for a property with the specified jsid*

function JS_HasUCProperty(cx: PJSContext; obj: PJSObject; const name: Pjschar; namelen: size_t; var found: JSBool): JSBool; cdecl; external SpiderMonkeyLib;

*Same as JS_HasProperty but unicode*

function JS_InitClass(cx: PJSContext; obj: PJSObject; parent_proto: PJSObject; clasp: PJSClass; _constructor: JSNative; nargs: uintN; ps: PJSPropertySpec; fs: PJSFunctionSpec; static_ps: PJSPropertySpec; static_fs: PJSFunctionSpec): PJSObject;

cdecl; external SpiderMonkeyLib;

*Make a JClass accessible to JavaScript code by creating its prototype, constructor, properties, and functions.*

- see https://developer.mozilla.org/en-US/docs/SpiderMonkey/JSAPI_Reference/JS_InitClass

function JS_InitStandardClasses(cx: PJSContext; obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;

*Extern JS_PUBLIC_API(JSObject *) JS_GetGlobalObject(JSContext *cx);

extern JS_PUBLIC_API(void) JS_SetGlobalObject(JSContext *cx, JSObject *obj); initializes the built-in JavaScript global properties.*

- These include all the standard ECMAScript global properties defined in ECMA 262-3 §15: Array, Boolean, Date, decodeURI, decodeURIComponent, encodeURI, encodeURIComponent, Error, eval, EvalError, Function, Infinity, isNaN, isFinite, Math, NaN, Number, Object, parseInt, parseFloat, RangeError, ReferenceError, RegExp, String, SyntaxError, TypeError, undefined, and URIError

- and set obj as a global object for cx!

function JS_InternJSString(cx: PJSContext; str: PJSString): PJSString; cdecl; external SpiderMonkeyLib;

*Get an interned string from a given null-terminated C string*

function JS_InternString(cx: PJSContext; s: PCChar): PJSString; cdecl; external SpiderMonkeyLib;

*Get an interned string from a given text buffer — that is, a JString that is protected from GC and automatically shared with other code that needs a JString with the same value*

- JS_InternUString and JS_InternUCStringN are the Unicode versions of the function.

- Each JSRuntime keeps a table of all existing interned strings. If an interned string already exists with the desired value, these functions return the existing string. Otherwise a new string is created and added to the table.

- Strings created with these functions are protected from garbage collection for the lifetime of the JSRuntime

- On success, these functions return a pointer to the interned string.

- Otherwise they report an error and return NULL.
function JS_InternUCStringN(cx: PJSContext; const s: pjschar; len: size_t): PJSString; cdecl; external Spider MonkeyLib;

Unencoded version of JS_InternString (faster)

function JS_IsAboutToBeFinalized(obj: PPJSObject): JSBool; cdecl; external Spider MonkeyLib;

JS_IsAboutToBeFinalized() checks if the given object is going to be finalized at the end of the current GC
- When called outside of the context of a GC, this function will return false
- Typically this function is used on weak references, where the reference should be nulled out or destroyed if the given object is about to be finalized
- The argument to JS_IsAboutToBeFinalized is an in-out param: when the function returns false, the object being referenced is still alive, but the garbage collector might have moved it. In this case, the reference passed to JS_IsAboutToBeFinalized will be updated to the object's new location
- Callers of this method are responsible for updating any state that is dependent on the object's address. For example, if the object's address is used as a key in a hashtable, then the object must be removed and re-inserted with the correct hash

function JS_IsArrayBufferObject(obj: PJSObject): JSBool; cdecl; external Spider MonkeyLib;

Check whether obj supports the JS_GetArrayBuffer* APIs
- Note that this may return false if a security wrapper is encountered that denies the unwrapping
- If this test succeeds, then it is safe to call the various accessor JSAPI calls

function JS_IsArrayBufferViewObject(obj: PJSObject): JSBool; cdecl; external Spider MonkeyLib;

Check whether obj supports JS_GetArrayBufferView* APIs
- Note that this may return false if a security wrapper is encountered that denies the unwrapping.
- If this test or one of the JS_Is*Array tests succeeds, then it is safe to call the dedicated ArrayBufferView accessor JSAPI calls

function JS_IsArrayObject(cx: PJSContext; obj: PJSObject): JSBool; cdecl; external Spider MonkeyLib;

Check if the supplied object is an array

function JS_IsConstructing(cx: PJSContext; const vp: PjsvalVector): boolean;

JS_isConstructing must be called from within a native given the native's original cx and vp arguments
- If JS_IsConstructing is true, JS_THIS must not be used; the constructor should construct and return a new object
- Otherwise, the native is called as an ordinary function and JS_THIS may be used

function JS_IsExceptionPending(cx: PJSContext): JSBool; cdecl; external Spider MonkeyLib;

Determine whether an exception is pending in the JS engine.
- JS_IsExceptionPending returns JS_TRUE if an exception has been thrown in the context cx and the exception has not yet been caught or cleared. Otherwise, it returns JS_FALSE
- This can be used from JSNative functions which call JS code to determine if the called JS code threw an exception or not.

function JS_IsExtensible(obj: PJSObject): JSBool; cdecl; external Spider MonkeyLib;

Queries the [[Extensible]] property of the object.
function JS_IsFloat32Array(obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;
Test for specific 32 bit float (single) typed array types (ArrayBufferView subtypes)

function JS_IsFloat64Array(obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;
Test for specific 64 bit float (double) typed array types (ArrayBufferView subtypes)

function JS_IsInRequest(cx: PJSContext): JSBool; cdecl; external SpiderMonkeyLib;

OBSOLETE procedure JS_YieldRequest(cx: PJSContext); cdecl; external SpiderMonkeyLib;
OBSOLETE type OBSOLETE   jsrefcount = JSInt32; OBSOLETE function JS_SuspendRequest(cx: PJSContext): jsrefcount; cdecl; external SpiderMonkeyLib;
OBSOLETE procedure JS_ResumeRequest(cx: PJSContext; saveDepth: jsrefcount); cdecl; external SpiderMonkeyLib; OBSOLETE

checked is is within a code region protected by JS_BeginRequest()

function JS_IsInt16Array(obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;
Test for specific 16 bit signed integer typed array types (ArrayBufferView subtypes)

function JS_IsInt32Array(obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;
Test for specific 32 bit signed integer typed array types (ArrayBufferView subtypes)

function JS_IsInt8Array(obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;
Test for specific 8 bit signed integer typed array types (ArrayBufferView subtypes)

function JS_IsRunning(cx: PJSContext): JSBool; cdecl; external SpiderMonkeyLib;
Determines if a script or function is currently executing in a specified JSContext, cx
- If a script is executing, JS_IsRunning returns JS_TRUE
- Otherwise it returns JS_FALSE.

function JS_IsTypedArrayObject(obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;
Check whether obj supports JS_GetTypedArray* APIs
- Note that this may return false if a security wrapper is encountered that denies the unwrapping.
- if this test or one of the JS_Is*Array tests succeeds, then it is safe to call the dedicated accessor JSAPI calls

function JS_IsUint16Array(obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;
Test for specific 16 bit unsigned integer typed array types (ArrayBufferView subtypes)

function JS_IsUint32Array(obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;
Test for specific 32 bit unsigned integer typed array types (ArrayBufferView subtypes)

function JS_IsUint8Array(obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;
Test for specific 8 bit unsigned integer typed array types (ArrayBufferView subtypes)

function JS_IsUint8ClampedArray(obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;
Test for specific 8 bit unsigned integer typed array types (ArrayBufferView subtypes)

procedure JS_LeaveCompartment(cx: PJSContext; oldCompartment: PJSCompartment); cdecl; external SpiderMonkeyLib;
Declare leaving the safe compartment of the specified object

function JS_LineNumberToPC(cx: PJSContext; script: PJSScript; lineno: uintn): pjsbytecode; cdecl; external SpiderMonkeyLib;
Retrieve the byte code item position corresponding to a script line number
function JS_LocalNameToAtom(w: JSuintptr): PJSAtom; cdecl; external SpiderMonkeyLib;

Convert a local name into a JavaScript atom

function JS_LookupElement(cx: PJSContext; obj: PJSObject; index: jsint; var vp: jsval): JSBool; cdecl; external SpiderMonkeyLib;

Determine if a specified numeric property exists
- examines a specified JavaScript object, obj, for a numeric property numbered index
- On success, *vp receives the stored value of the property, if any.

function JS_LookupProperty(cx: PJSContext; obj: PJSObject; name: PCChar; var vp: jsval): JSBool; cdecl; external SpiderMonkeyLib;

Determine if a specified property exists, according to its name
- On success, *vp receives the stored value of the property, if any.

function JS_LookupPropertyById(cx: PJSContext; obj: PJSObject; id: jsid; var vp: jsval): JSBool; cdecl; external SpiderMonkeyLib;

Determine if a specified property exists, according to its jsid

function JS_LookupUCProperty(cx: PJSContext; obj: PJSObject; const name: Pjschar; namelen: size_t; var vp: jsval): JSBool; cdecl; external SpiderMonkeyLib;

Determine if a specified property exists
- On success, *vp receives the stored value of the property, if any.

procedure JS_MaybeGC(cx: PJSContext); cdecl; external SpiderMonkeyLib;

Check if it would be worth it to launch the GarbageCollection process of the given execution RunTime, in the given context
- this function is a no-op if there is nothing interesting to garbage

function JS_New(cx: PJSContext; ctor: PJSObject; argc: uintN; argv: Pjsval): PJSObject; cdecl; external SpiderMonkeyLib;

As of SpiderMonkey 1.8.8, JS_ConstructObject and JS_ConstructObjectWithArguments have been removed from the JSAPI. The preferred alternative is to save a copy of the constructor function for the class, then to call it using JS_New. function JS_ConstructObject(cx: PJSContext; clasp: PJSClass; proto: PJSObject; parent: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib; function JS_ConstructObjectWithArguments(cx: PJSContext; clasp: PJSClass; proto: PJSObject; parent: PJSObject; argc: uintN; argv: pjsval): PJSObject; cdecl; external SpiderMonkeyLib; Create an object as though by using the new keyword and a JavaScript function
- for instance:
  JS_New(cx, ctor, argc, argv)

is equivalent to the JavaScript expression
  new ctor(argv[0], argv[1], ... argv[argc-1]).

If ctor is not an object that can be used as a constructor, a TypeError is raised.

function JS_NewArrayBuffer(cx: PJSContext; nbytes: uint32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new ArrayBuffer with the given byte length.
function JS_NewArrayObject(cx: PJSContext; length: jsint; vector: PjsvalVector): PJSObject; cdecl; external SpiderMonkeyLib;

JS_NewArrayObject creates a new array object with the specified length.
- If vector is non-null, then for each index i from 0 to length - 1,
  - JS_NewArrayObject defines an enumerable array element with the value vector[i] on the new array (This means that if length is nonzero and vector is null, the result is like the JavaScript expression new Array(length): i.e. the new array has the specified length, but it doesn't have any elements)
- On success, JS_NewArrayObject returns the new array object.
- Otherwise it reports an error as though by calling JS_ReportOutOfMemory and returns NULL.
- IMPORTANT! It is often better to call JS_NewArrayObject(cx, 0, NULL), store the returned object in a GC root, and then populate its elements with JS_SetElement or JS_DefineElement: this avoids unrooted jsvals in vector from being subject to garbage collection until the new object has been populated

function JS_NewContext(rt: PJSRuntime; stackChunkSize: size_t): PJSContext; cdecl; external SpiderMonkeyLib;

Create a new JSContext

function JS_NewDateObject(cx: PJSContext; year, mon, mday, hour, min, sec: int32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new JavaScript date object

function JS_NewDateObjectMsec(cx: PJSContext; msec: jsdouble): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new JavaScript date object from the Unix millisecond elapsed since EPOC

function JS_NewExternalString(cx: PJSContext; chars: pjschar; length: size_t; const fin: PJSStringFinalizer): PJSString; cdecl; external SpiderMonkeyLib;

Creates a new JSString whose characters are stored in external memory, i.e., memory allocated by the application, not the JavaScript engine
- Since the program allocated the memory, it will need to free it; this happens in an external string finalizer indicated by the type parameter.
- chars is Pointer to the first element of an array of jschars. This array is used as the character buffer of the JSString to be created. The array must be populated with the desired character data before JS_NewExternalString is called, and the array must remain in memory, with its contents unchanged, for as long as the JavaScript engine needs to hold on to it. (Ultimately, the string will be garbage collected, and the JavaScript engine will call the string finalizer callback, allowing the application to free the array)
- The text buffer array does not need to be zero-terminated.

function JS_NewFloat32Array(cx: PJSSContext; nelements: uint32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new signed 32 bit float (single) typed array with nelements elements
- will fill the newly created array with zeros

function JS_NewFloat32ArrayFromArray(cx: PJSContext; arr: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new 32 bit float (single) typed array and copy in values from a given object
- The object is used as if it was an array; that is, the new array (if successfully created) will have length given by array.length, and its elements will be those specified by array[0], array[1], and so on, after conversion to the typed array element type.
function JS_NewFloat32ArrayWithBuffer(cx: PJSContext; arrayBuffer: PJSObject; byteOffset: uint32; length: int32): PJSObject; cdecl; external SpiderMonkeyLib;

*Create a new 32 bit float (single) typed array using the given ArrayBuffer for storage*
- The length value is optional; if -1 is passed, enough elements to use up the remainder of the byte array is used as the default value

function JS_NewFloat64Array(cx: PJSContext; nelements: uint32): PJSObject; cdecl; external SpiderMonkeyLib;

*Create a new signed 64 bit float (double) typed array with nelements elements*
- will fill the newly created array with zeros

function JS_NewFloat64ArrayFromArray(cx: PJSContext; arr: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

*Create a new 64 bit float (double) typed array and copy in values from a given object*
- The object is used as if it was an array; that is, the new array (if successfully created) will have length given by array.length, and its elements will be those specified by array[0], array[1], and so on, after conversion to the typed array element type.

function JS_NewFloat64ArrayWithBuffer(cx: PJSContext; arrayBuffer: PJSObject; byteOffset: uint32; length: int32): PJSObject; cdecl; external SpiderMonkeyLib;

*Create a new 64 bit float (double) typed array using the given ArrayBuffer for storage*
- The length value is optional; if -1 is passed, enough elements to use up the remainder of the byte array is used as the default value

function JS_NewFunction(cx: PJSContext; call: JSNative; nargs: uintN; flags: uintN; parent: PJSObject; const name: PCChar): PJSFunction; cdecl; external SpiderMonkeyLib;

*JS_NewFunction creates a new JavaScript function implemented in Delphi*
- To create a new function implemented in JavaScript, use JS_CompileFunction
- call is a C/C++ function pointer that the new function wraps
- nargs is the number of arguments the function expects
- flags must be 0. parent may be used to specify the new function’s parent; NULL is usually the right thing here.
- name is the name to assign to the function. If name is NULL, the new function has no name. (JS_GetFunctionId, passed the new function, will return NULL)
- On success, JS_NewFunction returns a pointer to the newly created function
- Otherwise it reports an out-of-memory error and returns NULL

function JS_NewFunctionById(cx: PJSContext; call: JSNative; nargs: uintN; flags: uintN; parent: PJSObject; id: jsid): PJSFunction; cdecl; external SpiderMonkeyLib;

*Create the function with the name given by the id*
- JSID_IS_STRING(id) must be true.

function JS_NewGlobalObject(cx: PJSContext; clasp: PJSClass; principals: PJSPrincipals; options: PCompartmentOptions): PJSObject; cdecl; external SpiderMonkeyLib;

*JS_NewGlobalObject creates a new global object based on the specified class*
- The new object has no parent. It initially has no prototype either, since it is typically the first object created; call JS_InitStandardClasses to create all the standard objects, including Object.prototype, and set the global object’s prototype
function JS_NewInt16Array(cx: PJSContext; nelements: uint32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new signed 16 bit integer typed array with nelements elements
- will fill the newly created array with zeros

function JS_NewInt16ArrayFromArray(cx: PJSContext; arr: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new 16 bit signed integer typed array and copy in values from a given object
- The object is used as if it was an array; that is, the new array (if successfully created) will have length given by array.length, and its elements will be those specified by array[0], array[1], and so on, after conversion to the typed array element type.

function JS_NewInt16ArrayWithBuffer(cx: PJSContext; arrayBuffer: PJSObject; byteOffset: uint32; length: int32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new 16 bit signed integer typed array using the given ArrayBuffer for storage
- The length value is optional; if -1 is passed, enough elements to use up the remainder of the byte array is used as the default value

function JS_NewInt32Array(cx: PJSContext; nelements: uint32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new signed 32 bit integer typed array with nelements elements
- will fill the newly created array with zeros

function JS_NewInt32ArrayFromArray(cx: PJSContext; arr: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new 32 bit signed integer typed array and copy in values from a given object
- The object is used as if it was an array; that is, the new array (if successfully created) will have length given by array.length, and its elements will be those specified by array[0], array[1], and so on, after conversion to the typed array element type.

function JS_NewInt32ArrayWithBuffer(cx: PJSContext; arrayBuffer: PJSObject; byteOffset: uint32; length: int32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new 32 bit signed integer typed array using the given ArrayBuffer for storage
- The length value is optional; if -1 is passed, enough elements to use up the remainder of the byte array is used as the default value

function JS_NewInt8Array(cx: PJSContext; nelements: uint32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new signed 8 bit integer typed array with nelements elements
- will fill the newly created array with zeros

function JS_NewInt8ArrayFromArray(cx: PJSContext; arr: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new 8 bit signed integer typed array and copy in values from a given object
- The object is used as if it was an array; that is, the new array (if successfully created) will have length given by array.length, and its elements will be those specified by array[0], array[1], and so on, after conversion to the typed array element type.

function JS_NewInt8ArrayWithBuffer(cx: PJSContext; arrayBuffer: PJSObject; byteOffset: uint32; length: int32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new 8 bit signed integer typed array using the given ArrayBuffer for storage
- The length value is optional; if -1 is passed, enough elements to use up the remainder of the byte array is used as the default value
function JS_NewObject(cx: PJSContext; clasp: PJSClass; proto: PJSObject; parent: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

`JS_NewObject` creates a new object based on a specified class, prototype, and parent object
- `cx` is a pointer to a context associated with the runtime in which to establish the new object.
- `clasp` is a pointer to an existing class to use for internal methods, such as finalize
- `proto` is an optional pointer to the prototype object with which to associate the new object see https://developer.mozilla.org/en-US/docs/SpiderMonkey/JSAPI_Reference/JS_NewObject

function JS_NewObjectWithGivenProto(cx: PJSContext; clasp: PJSClass; proto: PJSObject; parent: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

Unlike `JS_NewObject`, `JS_NewObjectWithGivenProto` does not compute a default proto if proto's actual parameter value is null

function JS_NewPropertyIterator(cx: PJSContext; obj: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

Create an object to iterate over enumerable properties of `obj`, in arbitrary
- property definition order. NB: This differs from longstanding for..in loop
- order, which uses order of property definition in `obj`.

function JS_NewRuntime(maxbytes: uint32; useHelperThreads: JSUseHelperThreads): PJSRuntime; cdecl; external SpiderMonkeyLib;

Initialize the JavaScript runtime
- `JS_NewRuntime` initializes the JavaScript runtime environment. Call `JS_NewRuntime` before making any other API calls.
- `JS_NewRuntime` allocates memory for the JSRuntime and initializes certain internal runtime structures. `maxbytes` specifies the number of allocated bytes after which garbage collection is run.

function JS_NewStringCopyN(cx: PJSContext; s: PCChar; n: size_t): PJSString; cdecl; external SpiderMonkeyLib;

Caller. So the caller must free bytes in the error case, if it has no use for them. In contrast, all the `JS_New//StringCopy//` functions do not take ownership of the character memory passed to them -- they copy it. `JS_NewStringCopyN` allocates space for a JavaScript string and its underlying storage, and copies `n` characters from a C character array, `s`, into the new JSString.
- `JS_NewUCStringCopyN` is the Unicode version of the function. The two functions differ only in the type of the character array `s`; both functions create ordinary JavaScript strings, and all JavaScript strings are made up of 16-bit characters.
- If the array `s` contains more than `n` characters, the new string contains a truncated version of the original string. The string may contain null characters (#0). They are copied into the new string like any other character.
- You can use `JS_NewStringCopyN` to copy binary data or to copy only a certain portion of a C string into a JavaScript string.
- On success, `JS_NewStringCopyN` and `JS_NewUCStringCopyN` return a pointer to the new JS string.
- Otherwise they return NULL.

function JS_NewStringCopyZ(cx: PJSContext; s: PCChar): PJSString; cdecl; external SpiderMonkeyLib;

Create a new JavaScript string based on a null-terminated C string
function JS_NewUCString(cx: PJSContext; s: pjschar; len: size_t): PJSString; cdecl; external SpiderMonkeyLib;

*Creates and returns a new string, using the memory starting at buf and ending at buf + length as the character storage.*
- The character array, buf, MUST be allocated on the heap using JS_malloc.
- On success, the JavaScript engine adopts responsibility for memory management of this region.
- The application must not read, write, or free the buffer.
- This allows the JavaScript engine to avoid needless data copying.
- On success, JS_NewUCString return a pointer to the new string.
- On error or exception, they return NUL
- !!!!! DO NOT USE THIS FUNCTION WITH DELPHI strings!!!!!!!

function JS_NewUCStringCopyN(cx: PJSContext; s: pjschar; n: size_t): PJSString; cdecl; external SpiderMonkeyLib;

*Unicode version of JS_NewStringCopyN*
- faster then JS_NewStringCopyN because no need to transform into unicode

function JS_NewUCStringCopyZ(cx: PJSContext; const s: pjschar): PJSString; cdecl; external SpiderMonkeyLib;

*Unicode version of JS_NewStringCopyZ*

function JS_NewUint16Array(cx: PJSContext; nelements: uint32): PJSObject; cdecl; external SpiderMonkeyLib;

*Create a new unsigned 16 bit integer typed array with nelements elements*
- will fill the newly created array with zeros

function JS_NewUint16ArrayFromArray(cx: PJSContext; arr: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

*Create a new 16 bit unsigned integer typed array and copy in values from a given object*
- The object is used as if it was an array; that is, the new array (if successfully created) will have length given by array.length, and its elements will be those specified by array[0], array[1], and so on, after conversion to the typed array element type.

function JS_NewUint16ArrayWithBuffer(cx: PJSContext; arrayBuffer: PJSObject; byteOffset: uint32; length: int32): PJSObject; cdecl; external SpiderMonkeyLib;

*Create a new 16 bit unsigned integer typed array using the given ArrayBuffer for storage*
- The length value is optional; if -1 is passed, enough elements to use up the remainder of the byte array is used as the default value

function JS_NewUint32Array(cx: PJSContext; nelements: uint32): PJSObject; cdecl; external SpiderMonkeyLib;

*Create a new unsigned 32 bit integer typed array with nelements elements*
- will fill the newly created array with zeros

function JS_NewUint32ArrayFromArray(cx: PJSContext; arr: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

*Create a new 32 bit unsigned integer typed array and copy in values from a given object*
- The object is used as if it was an array; that is, the new array (if successfully created) will have length given by array.length, and its elements will be those specified by array[0], array[1], and so on, after conversion to the typed array element type.
function JS_NewUint32ArrayWithBuffer(cx: PJSContext; arrayBuffer: PJSObject; byteOffset: uint32; length: int32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new 32 bit unsigned integer typed array using the given ArrayBuffer for storage
- The length value is optional; if -1 is passed, enough elements to use up the remainder of the byte array is used as the default value

function JS_NewUint8Array(cx: PJSContext; nelements: uint32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new unsigned 8 bit integer (byte) typed array with nelements elements
- will fill the newly created array with zeros

function JS_NewUint8ArrayFromArray(cx: PJSContext; arr: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new 8 bit unsigned integer typed array and copy in values from a given object
- The object is used as if it was an array; that is, the new array (if successfully created) will have length given by array.length, and its elements will be those specified by array[0], array[1], and so on, after conversion to the typed array element type.

function JS_NewUint8ArrayWithBuffer(cx: PJSContext; arrayBuffer: PJSObject; byteOffset: uint32; length: int32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new 8 bit unsigned integer typed array using the given ArrayBuffer for storage
- The length value is optional; if -1 is passed, enough elements to use up the remainder of the byte array is used as the default value

function JS_NewUint8ClampedArray(cx: PJSContext; nelements: uint32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new 8 bit integer typed array with nelements elements
- will fill the newly created array with zeros

function JS_NewUint8ClampedArrayFromArray(cx: PJSContext; arr: PJSObject): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new 8 bit unsigned integer typed array and copy in values from a given object
- The object is used as if it was an array; that is, the new array (if successfully created) will have length given by array.length, and its elements will be those specified by array[0], array[1], and so on, after conversion to the typed array element type.

function JS_NewUint8ClampedArrayWithBuffer(cx: PJSContext; arrayBuffer: PJSObject; byteOffset: uint32; length: int32): PJSObject; cdecl; external SpiderMonkeyLib;

Create a new 8 bit unsigned integer typed array using the given ArrayBuffer for storage
- The length value is optional; if -1 is passed, enough elements to use up the remainder of the byte array is used as the default value

function JS_NextProperty(cx: PJSContext; iterobj: PJSObject; var idp: jsid): JSBool; cdecl; external SpiderMonkeyLib;

Return true on success with *idp containing the id of the next enumerable
- property to visit using iterobj, or JSID_IS_VOID if there is no such property
- left to visit. Return false on error.

function JS_NumberValue(d: double): jsval; cdecl; external SpiderMonkeyLib name 'JS_NumberValue_';

JS_NewNumberValue Obsolete since JavaScript mozjs17 use JS_NumberValue instead
function JS_ObjectIsCallable(cx: PJSContext; obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;
  
  Check is the supplied object is callable

function JS_ObjectIsDate(cx: PJSContext; obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;
  
  Infallible predicate to test whether obj is a JavaScript date object

function JS_ObjectIsFunction(cx: PJSContext; obj: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;
  
  Test whether a given object is a Function

function JS_ParseJSON(cx: PJSContext; const chars: Pjschar; len: uint32; vp: Pjsval): JSBool; cdecl; external SpiderMonkeyLib;
  
  Parse a string using the JSON syntax described in ECMAScript 5 and return the corresponding value into vp

function JS_PCToLineNumber(cx: PJSContext; script: PJSScript; pc: pjsbytecode): uintn; cdecl; external SpiderMonkeyLib;
  
  Retrieve the script line number from a byte code item position

function JS_PropertyStub(cx: PJSContext; var obj: PJSObject; var id: jsid; vp: pjsval): JSBool; cdecl; external SpiderMonkeyLib;
  
  Default callback matching JSPropertyOp prototype of JSClass

procedure JS_PutPropertyDescArray(cx: PJSContext; var pda: JSPropertyDescArray); cdecl; external SpiderMonkeyLib;
  
  Define the description of a given JavaScript object property

procedure JS_ReleaseFunctionLocalNameArray(cx: PJSContext; markp: pointer); cdecl; external SpiderMonkeyLib;
  
  Release the local name array information of a given function

function JS_RemoveObjectRoot(cx: PJSContext; rp: PPJSObject): JSBool; cdecl; external SpiderMonkeyLib;
  
  Remove a JSObject variable from the garbage collector’s root set

function JS_RemoveStringRoot(cx: PJSContext; rp: PPJSString): JSBool; cdecl; external SpiderMonkeyLib;
  
  Remove a JSString variable from the garbage collector’s root set

function JS_RemoveValueRoot(cx: PJSContext; vp: Pjsval): JSBool; cdecl; external SpiderMonkeyLib;
  
  Remove a jsval variable from the garbage collector’s root set
procedure JS_ReportAllocationOverflow(cx: PJSContext); cdecl; external SpiderMonkeyLib;

Call JS_ReportAllocationOverflow if an operation fails because it tries to use more memory (or more of some other resource) than the application is designed to handle
- When a script tries to grow an array beyond 230-1 elements, for example, or concatenate strings such that the result is more than 229-1 characters long, the JavaScript engine reports an error as though by calling this function.
- The main difference between these two functions is that JS_ReportOutOfMemory does not cause a JavaScript exception to be thrown. The error therefore cannot be caught by try...catch statements in scripts. JS_ReportAllocationOverflow throws an InternalError which scripts can catch.

procedure JS_ReportError(cx: PJSContext; const format: PCChar); cdecl; varargs;
external SpiderMonkeyLib;

JS_ReportError is the simplest JSAPI function for reporting errors
- First it builds an error message from the given sprintf-style format string and any additional arguments passed after it. The resulting error message is passed to the context's JSErrorReporter callback, if any
- If the caller is in a JSAPI callback, JS_ReportError also creates a new JavaScript Error object and sets it to be the pending exception on cx
- The callback must then return JS_FALSE to cause the exception to be propagated to the calling script
- An example is shown in the JSAPI Phrasebook
- For internationalization, use JS_ReportErrorNumber instead
- To report an out-of-memory error, use JS_ReportOutOfMemory

procedure JS_ReportErrorNumber(cx: PJSContext; errorCallback: JSErrorCallback; userRef: pointer; const errorNumber: uint); cdecl; varargs; external SpiderMonkeyLib;

Report an error with an application-defined error code.
- varargs is Additional arguments for the error message.
- These arguments must be of type char*
- The number of additional arguments required depends on the error message, which is determined by the errorCallback

procedure JS_ReportErrorNumberUC(cx: PJSContext; errorCallback: JSErrorCallback; userRef: pointer; const errorNumber: uintN); cdecl; varargs; external SpiderMonkeyLib;

Report an error with an application-defined error code.
- varargs is Additional arguments for the error message.
- These arguments must be of type jschar*
- The number of additional arguments required depends on the error message, which is determined by the errorCallback

procedure JS_ReportOutOfMemory(cx: PJSContext); cdecl; external SpiderMonkeyLib;

Reports a memory allocation error
- Call JS_ReportOutOfMemory to report that an operation failed because the system is out of memory
- When the JavaScript engine tries to allocate memory and allocation fails, it reports an error as though by calling this function
procedure JS_ReportWarning(cx: PJSCxt; const format: PCChar); cdecl; varargs;
external SpiderMonkeyLib;

Similar to JS_ReportError(), but report a warning instead of an error
(JSREPORT_IS_WARNING(report.flags))
- Return true if there was no error trying to issue the warning, and if the warning was not
converted into an error due to the JSOPTION_WERROR option being set, false otherwise

function JS_ResolveStandardClass(cx: PJSCxt; obj: PJSOBJect; id: jsid; var resolved: JSBool): JSBool; cdecl; external SpiderMonkeyLib;

Resolve id, which must contain either a string or an int, to a standard class name in obj if possible,
defining the class's constructor and/or prototype and storing true in resolved
- If id does not name a standard class or a top-level property induced by initializing a standard
class, store false in *resolved and just return true.  Return false on error, as usual for JSBool
result-typed API entry points.
- This API can be called directly from a global object class's resolve op, to define standard classes
lazily.  The class's enumerate op should call JS_EnumerateStandardClasses(cx, obj), to define
eagerly during for..in loops any classes not yet resolved lazily.

function JS_ResolveStub(cx: PJSCxt; var obj: PJSOBJect; var id: jsid): JSBool; cdecl; external SpiderMonkeyLib;

Default callback matching JSResolveOp prototype of JSClass

procedure JS_RestoreExceptionState(cx: PJSCxt; state: PJSEXceptionState); cdecl;
external SpiderMonkeyLib;

Restore a specified exception state

function JS_RVAL(cx: PJSCxt; vp: Pjsval): jsval;
Get the return value of a JSNative callback

function JS_SameValue(cx: PJSCxt; v1, v2: jsval; equal: PJSBool): JSBool; cdecl;
external SpiderMonkeyLib;

Determines if two jsvals are the same, as determined by the SameValue algorithm in ECMAScript
262, 5th edition
- SameValue slightly differs from strict equality (===) in that +0 and -0 are not the same and in that
NaN is the same as NaN

function JS_SaveExceptionState(cx: PJSCxt): PJSEXceptionState; cdecl; external SpiderMonkeyLib;

Save the current exception state.  This takes a snapshot of cx’s current exception state without
making any change to that state.
- The returned state pointer MUST be passed later to JS_RestoreExceptionState (to restore that
saved state, overriding any more recent state) or else to JS_DropExceptionState (to free the state
struct in case it is not correct or desirable to restore it).  Both Restore and Drop free the state
struct, so callers must stop using the pointer returned from Save after calling the Release or Drop
API.
function JS_SetArrayLength(cx: PJSContext; obj: PJSObject; length: jsuint): JSBool;
cdecl; external SpiderMonkeyLib;

JS_SetArrayLength sets the length property of an object obj
- length indicates the number of elements
- JS_SetArrayLength(cx, obj, n) is exactly the same as setting the length property of obj to n using
  JS_SetProperty
- This is true even if obj is not an Array object
- You can call JS_SetArrayLength either to set the number of elements for an array object you
  created without specifying an initial number of elements, or to change the number of elements
  allocated for an array
- If you set a shorter array length on an existing array, the elements that no longer fit in the array
  are destroyed
- Setting the number of array elements does not initialize those elements
- To initialize an element call JS_DefineElement
- If you call JS_SetArrayLength on an existing array, and length is less than the highest index
  number for previously defined elements, all elements greater than or equal to length are
  automatically deleted
- On success, JS_SetArrayLength returns JS_TRUE. Otherwise it returns JS_FALSE.

function JS_SetContextCallback(rt: PJSRuntime; cxCallback: JSContextCallback):
JSContextCallback;
cdecl; external SpiderMonkeyLib;

Specifies a callback function that is automatically called whenever a JSContext is created or
destroyed

procedure JS_SetContextPrivate(cx: PJSContext; data: Pointer);
cdecl; external SpiderMonkeyLib;

Write access to a JSContext field for application-specific data

function JS_SetDebugErrorHook(rt: PJSRuntime; hook: JSDebugErrorHook; closure:
pointer): JSBool;
cdecl; external SpiderMonkeyLib;

Set a callback to be called when an error is triggered during script debugging

function JS_SetDebuggerHandler(rt: PJSRuntime; hook: JSDebuggerHandler; closure:
pointer): JSBool;
cdecl; external SpiderMonkeyLib;

Set a debugging handler callback for a given hook and runtime

function JS_SetDebugMode(cx: PJSContext; debug: JSBool): JSBool;
cdecl; external SpiderMonkeyLib;

Enables the JavaScript debugging mode for a given context

procedure JS_SetDestroyScriptHookProc(rt: PJSRuntime; hook: JSDestroyScriptHook;
callerdata: Pointer);
cdecl; external SpiderMonkeyLib;

Set the callback to be called when a JavaScript debugging hook is released

function JS_SetElement(cx: PJSContext; obj: PJSObject; index: jsint; var vp: jsval):
JSBool;
cdecl; external SpiderMonkeyLib;

Assign a value to a numeric property of an object
**function** JS_SetErrorReporter(cx: PJSContext; er: JSErrorReporter): JSErrorReporter;
**cdecl; external** SpiderMonkeyLib;

*Specify the error reporting mechanism for an application.*
- JS_SetErrorReporter enables you to define and use your own error reporting mechanism in your applications.
- The reporter you define is automatically passed a JSErrorReport structure when an error occurs and has been parsed by JS_ReportError.
- JS_SetErrorReporter returns the previous error reporting function of the context, or NULL if no such function had been set.
- Typically, the error reporting mechanism you define should log the error where appropriate (such as to a log file), and display an error to the user of your application.
- The error you log and display can make use of the information passed about the error condition in the JSErrorReport structure.
- The error reporter callback must not reenter the JSAPI.
- Like all other SpiderMonkey callbacks, the error reporter callback must not throw any Delphi exception.

**function** JS_SetGCCallback(rt: PJSRuntime; cb: JSGCCallback): JSGCCallback;
**cdecl; external** SpiderMonkeyLib;

*Defines a Garbage Collection call-back function*

**function** JS_SetInterrupt(rt: PJSRuntime; handler: JSInterruptHook; closure: pointer): JSBool;
**cdecl; external** SpiderMonkeyLib;

*Set a callback to be called when script debugging is interrupted*

**procedure** JS_SetNativeStackQuota(tr: PJSRuntime; stackSize: size_t);
**cdecl; external** SpiderMonkeyLib;

*Set the size of the native stack that should not be exceeded*
- to disable stack size checking, just pass 0 as value

**procedure** JS_SetNewScriptHookProc(rt: PJSRuntime; hook: JSNewScriptHook; callerdata: Pointer);
**cdecl; external** SpiderMonkeyLib;

*Set a JavaScript debugging hook callback*
**function** JS_SetOperationCallback(cx: PJSContext; callback: JSOperationCallback): JSOperationCallback; cdecl; external SpiderMonkeyLib;

These functions allow setting an operation callback that will be called from the thread the context is associated with some time after any thread triggered the callback using JS_TriggerOperationCallback(cx).

In a threadsafe build the engine internally triggers operation callbacks under certain circumstances (i.e. GC and title transfer) to force the context to yield its current request, which the engine always automatically does immediately prior to calling the callback function. The embedding should thus not rely on callbacks being triggered through the external API only.

Important note: Additional callbacks can occur inside the callback handler if it re-enters the JS engine. The embedding must ensure that the callback is disconnected before attempting such re-entry. Set a callback function that is automatically called periodically while JavaScript code runs:
- cx is a Pointer to a JSContext in which this callback was installed.
- The callback may use this context to call JSAPI functions, but it should first use JS_SetOperationCallback
- to set the context's operation callback to NULL. Otherwise the engine may call the operation callback again, reentering it.
- Provides request. In JS_THREADSAFE builds, the JavaScript engine calls this callback only from within an active request on cx.
- The callback does not need to call JS_BeginRequest()
- Some common uses for an operation callback are: To run garbage collection periodically, by calling JS_MaybeGC; To periodically take a break from script execution to update the UI (though note that Mozilla does not do this, by design); To enforce application limits on the amount of time a script may run. (In this case, the callback may terminate the script by returning JS_FALSE.)

**procedure** JS_SetPendingException(cx: PJSContext; v: jsval); cdecl; external SpiderMonkeyLib;

Set the current exception being thrown within a context
- JS_SetPendingException sets the current exception being thrown within a context. If an exception is already being thrown, it is replaced with the new one given.
- v is the new value to throw as an exception.
- A native function or hook using this to throw an exception must also return JS_FALSE to ensure the exception is thrown.
- Each JSContext's pending-exception field is a GC root. That is, garbage collection never collects a pending exception

**function** JS_SetProperty(cx: PJSContext; obj: PJSObject; const name: PCChar; var vp: jsval): JSBool; cdecl; external SpiderMonkeyLib;

The following functions behave like JS_GetProperty and JS_GetPropertyById except when operating on E4X XML objects extern JS_PUBLIC_API(JSBool) JS_GetMethodById(JSContext *cx, JSObject *obj, jsid id, JSObject **objp, jsval *vp); extern JS_PUBLIC_API(JSBool) JS_GetMethod(JsContext *cx, JSObject *obj, const char *name, JSObject **objp, jsval *vp);
JS_SetProperty assigns the value vp to the property name of the object obj
- it behaves like the JavaScript expression obj[name] = v
- it will create the property if it does not exist, but for details see https://developer.mozilla.org/en-US/docs/SpiderMonkey/JSAPI_Reference/JS_SetProperty
- remark: in mozilla description "sealed" object they talking about is OBSOLETE!
function JS_SetPropertyAttributes(cx: PJSContext; obj: PJSObject; const name: PCChar; attr: uintN; var foundp: JSBool): JSBool; cdecl; external SpiderMonkeyLib;
  
  Set the attributes of a property on a given object.
  - If the object does not have a property by that name, *foundp will be JS_FALSE and nothing will be altered.

function JS_SetPropertyById(cx: PJSContext; obj: PJSObject; id: jsid; vp: Pjsval): JSBool; cdecl; external SpiderMonkeyLib;
  
  JS_SetProperty assigns the value vp to the property jsid of the object obj.

function JS_SetReservedSlot(cx: PJSContext; obj: PJSObject; index: uint32; v: jsval): JSBool; cdecl; external SpiderMonkeyLib;
  
  Writes access an object's reserved slots.

procedure JS_SetRuntimeDebugMode(rt: PJSRuntime; debug: JSBool); cdecl; external SpiderMonkeyLib;
  
  Enables the JavaScript debugging mode for a given runtime.

function JS_SetSingleStepMode(cx: PJSContext; script: PJSScript; singleStep: JSBool): JSBool; cdecl; external SpiderMonkeyLib;
  
  Set single step mode. In this mode script interrupts on each line.

function JS_SetThrowHook(rt: PJSRuntime; hook: JSThrowHook; closure: pointer): JSBool; cdecl; external SpiderMonkeyLib;
  
  Set a callback to be called when script throws an exception.

function JS_SetTrap(cx: PJSContext; script: PJSScript; pc: pjsbytecode; handler: JSTrapHandler; closure: jsval): JSBool; cdecl; external SpiderMonkeyLib;
  
  Set a trap debugging handler callback for a given execution context.

function JS_SetUCPropertyAttributes(cx: PJSContext; obj: PJSObject; const name: Pjschar; namelen: size_t; attr: uintN; var foundp: JSBool): JSBool; cdecl; external SpiderMonkeyLib;
  
  Set the attributes of a property on a given object.
  - If the object does not have a property by that name, *foundp will be
  - JS_FALSE and nothing will be altered.

procedure JS_SET_RVAL(cx: PJSContext; vp: Pjsval; v: jsval);
  
  Set the return value of a JSNative callback.

procedure JS_ShutDown;
  
  Free all resources used by the JS engine, not associated with specific runtimes.

function JS_StrictlyEqual(cx: PJSContext; v1, v2: jsval; equal: PJSBool): JSBool; cdecl; external SpiderMonkeyLib;
  
  Determine whether two JavaScript values are equal in the sense of the === operator.

function JS_StrictPropertyStub(cx: PJSContext; var obj: PJSObject; var id: jsid; _strict: JSBool; vp: Pjsval): JSBool; cdecl; external SpiderMonkeyLib;
  
  Default callback matching JSStrictPropertyOp prototype of JClass.

function JS_Stringify(cx: PJSContext; vp: Pjsval; replacer: PJSObject; space: jsval; callback: JSONWriteCallback; data: pointer): JSBool; cdecl; external SpiderMonkeyLib;
  
  Converts a value to JSON, optionally replacing values if a replacer function is specified, or
  optionally including only the specified properties if a replacer array is specified.
function JS_StringToVersion(_string: PCChar): JSVersion; cdecl; external SpiderMonkeyLib;
    // Configure a JSContext to use a specific version of the JavaScript language

function JS_THIS(cx: PJSContext; vp: Pjsval): jsval;
    // Return the this object of a JSNative callback

function JS_THIS_OBJECT(cx: PJSContext; vp: Pjsval): PJSObject;
    // Return the this object of a JSNative callback

function JS_ThrowReportedError(cx: PJSContext; const msg: PCChar; reportp: PJSErrorReport): JSBool; cdecl; external SpiderMonkeyLib;
    // Given a reported error's message and JSErrorReport struct pointer, throw the corresponding exception on cx

function JS_ThrowStopIteration(cx: PJSContext): JSBool; cdecl; external SpiderMonkeyLib;
    // Throws a StopIteration exception on cx

procedure JS_TriggerOperationCallback(rt: PJSRuntime); cdecl; external SpiderMonkeyLib;
    // Triggers a callback set using JS_SetOperationCallback

function JS_TypeOfValue(cx: PJSContext; v: jsval): JSType; cdecl; external SpiderMonkeyLib;
    // Determines the JS data type of a JS value

function JS_ValueToBoolean(cx: PJSContext; v: jsval; var b: JSBool): JSBool; cdecl; external SpiderMonkeyLib;
    // Convert a JavaScript value to a boolean

function JS_ValueToECMAInt32(cx: PJSContext; v: jsval; var i: int32): JSBool; cdecl; external SpiderMonkeyLib;
    // Convert a JavaScript value to an integer type as specified by the ECMAScript standard

function JS_ValueToECMAUint32(cx: PJSContext; v: jsval; var ui: uint32): JSBool; cdecl; external SpiderMonkeyLib;
    // Convert a JavaScript value to an integer type as specified by the ECMAScript standard

function JS_ValueToFunction(cx: PJSContext; v: jsval): PJSFunction; cdecl; external SpiderMonkeyLib;
    // Convert a jsval to a JSFunction
    // - If v is a Function object, this returns the associated JSFunction.
    // - If v is null, undefined, a boolean, a number, or a string, a TypeError is reported and JS_ValueToFunction returns nil
    // - Otherwise, v is an object. The JavaScript engine attempts to convert it to a function, which can be pretty unsafe! You have been warned!

function JS_ValueToInt32(cx: PJSContext; v: jsval; var i: int32): JSBool; cdecl; external SpiderMonkeyLib;
    // JS_ValueToInt32 is obsolete

function JS_ValueToNumber(cx: PJSContext; v: jsval; var dp: jsdouble): JSBool; cdecl; external SpiderMonkeyLib;
    // Convert any JavaScript value to a floating-point number of type jsdouble.
function JS_ValueToObject(cx: PJSContext; v: jsval; var objp: PJSObject): JSBool; cdecl; external SpiderMonkeyLib;

*JS_ValueToObject converts a specified JavaScript value, v, to an object*
- On success, this function stores either NULL or a pointer to the resulting object in objp and returns JS_TRUE. Otherwise it returns JS_FALSE and the value left in objp is unspecified.
- If v is JSVAL_NULL or JSVAL_VOID, the result is NULL.
- If v is a boolean value, a number, or a string, the result is a new wrapper object of type Boolean, Number, or String.
- Otherwise v is an object, and the result depends on the object. If v is a native JavaScript Object, this calls the object's valueOf method, if any. In any case, the result is not guaranteed to be the same object as v. (Implementation note: the object's JSObjectOps.defaultValue method is called with hint=JSTYPE_OBJECT.)
- The resulting object is subject to garbage collection unless the variable objp is protected by a local root scope, an object property, or the JS_AddRoot function. Note that a local root scope is not sufficient to protect the resulting object in some cases involving the valueOf method!

function JS_ValueToSource(cx: PJSContext; v: jsval): PJSString; cdecl; external SpiderMonkeyLib;

*Convert any JavaScript value to its source representation*

function JS_ValueToString(cx: PJSContext; v: jsval): PJSString; cdecl; external SpiderMonkeyLib;

*JS_ValueToString converts a specified JavaScript value, v, to a string.*
- It implements the ToString operator specified in ECMA 262-3 §9.8
- The result is like the JavaScript expression ""+v.
- If v is already a string, conversion succeeds.
- If v is true, false, null, or undefined, conversion succeeds, and the result is the string "true", "false", "null", or "undefined", accordingly.
- If v is a number, conversion succeeds, and the result is a string representation of that number as specified in ECMA 262-3 §9.8.1. This might be "NaN", "Infinity", or "-Infinity". Otherwise the result is a decimal representation of the number, possibly using exponential notation.
- Otherwise v is an object. JS_ValueToString uses the steps below to convert it to a string. If at any point an error or exception occurs, or conversion succeeds, the rest of the steps are skipped. (This behavior is implemented by v's JSObjectOps.defaultValue method, so host objects can override it all.) If v.toString() is a function, it is called. If that method returns a primitive value, the value is converted to a string as described above and conversion succeeds. Otherwise, the resulting object's JSClass.convert callback is called. For standard classes, this is JS_ConvertStub, which simply calls v.valueOf() if present. If the convert callback produces a primitive value, the value is converted to a string as described above and conversion succeeds. Otherwise conversion fails with a TypeError.
- On success, JS_ValueToString returns a pointer to a string. On error or exception, it returns NULL. This happens, for example, if v is an object and v.toString() throws an exception.
- The resulting JSString is subject to garbage collection. Protect it using a local root, an object property, or the JS_AddRoot function.

function JS_ValueToUint16(cx: PJSContext; v: jsval; var ui16: uint16): JSBool; cdecl; external SpiderMonkeyLib;

*Convert a JavaScript value to a 16 bit integer*
- ECMA ToUint16, e.g. for mapping a jsval to a Unicode point.
function JS_VersionToString(version: JSVersion): PCChar; cdecl; external SpiderMonkeyLib;
  
  JS_SetVersion is not supported now Use CompartmentOptions in JS_NewGlobalObject retrieve the
  JavaScript version text used within a specified executable script context

function PR_CreateThread( type_: PRThreadType; start: pointer; arg: pointer;
priority: PRThreadPriority; scope: PRThreadScope; state: PRThreadState; stackSize: PRUint32): PRThread; cdecl; external NSPRLib;
  Initializes a NSPR thread

procedure PR_DestroyCondVar(cvar: PRCondVar); cdecl; external NSPRLib;
  Free a previously allocated NSPR event

procedure PR_DestroyLock(lock: PRLock); cdecl; external NSPRLib;
  Free a previously allocated NSPR lock

function PR_JoinThread(thread: PRThread): PRStatus; cdecl; external NSPRLib;
  Join a NSPR thread

procedure PR_Lock(lock: PRLock); cdecl; external NSPRLib;
  Enter a previously allocated NSPR mutex/lock

function PR_NewCondVar(lock: PRLock): PRCondVar; cdecl; external NSPRLib;
  Allocates a new NSPR event

function PR_NewLock: PRLock; cdecl; external NSPRLib;
  Allocates a new NSPR mutex/lock

function PR_NotifyAllCondVar(cvar: PRCondVar): PRStatus; cdecl; external NSPRLib;
  Notify all previously allocated NSPR event

function PR_NotifyCondVar(cvar: PRCondVar): PRStatus; cdecl; external NSPRLib;
  Notify a previously allocated NSPR event

function PR_SetCurrentThreadName(name: PAnsiChar): PRStatus; cdecl; external NSPRLib;
  Change the current NSPR thread name

function PR_TicksPerSecond(): PRUint32; cdecl; external NSPRLib;
  Returns the number of ticks per seconds as expected by NSPR

function PR_Unlock(lock: PRLock): PRStatus; cdecl; external NSPRLib;
  Leave a previously allocated NSPR mutex/lock

function PR_WaitCondVar(cvar: PRCondVar; timeout: PRIntervalTime): PRStatus; cdecl;
external NSPRLib;
  Wait until a previously allocated NSPR event is notified
27.35. **SynSQLite3.pas unit**

*Purpose*: SQLite3 Database engine direct access
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The **SynSQLite3** unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.2.1</td>
<td>The SQLite3 engine shall be embedded to the framework</td>
<td>2548</td>
</tr>
</tbody>
</table>

**Units used in the SynSQLite3 unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/cache/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

**SynSQLite3 class hierarchy**

**Objects implemented in the SynSQLite3 unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td><code>ESQLite3Exception</code></td>
<td>Custom SQLite3 dedicated Exception type</td>
<td>1678</td>
</tr>
<tr>
<td><code>TFTSMatchInfo</code></td>
<td>Map the matchinfo function returned BLOB value</td>
<td>1647</td>
</tr>
<tr>
<td><code>TSQLBlobStream</code></td>
<td>Used to read or write a BLOB Incrementalaly</td>
<td>1695</td>
</tr>
<tr>
<td><code>TSQLDataBase</code></td>
<td>Simple wrapper for direct SQLite3 database manipulation</td>
<td>1686</td>
</tr>
<tr>
<td><code>TSQLDatabaseBackupThread</code></td>
<td>Background thread used for TSQLDatabase.BackupBackground() process</td>
<td>1695</td>
</tr>
<tr>
<td><code>TSQLDataBaseSQLFunction</code></td>
<td>Those classes can be used to define custom SQL functions inside a TSQLDatabase</td>
<td>1685</td>
</tr>
<tr>
<td><code>TSQLDataBaseSQLFunctionDynArray</code></td>
<td>To be used to define custom SQL functions for dynamic arrays BLOB search</td>
<td>1685</td>
</tr>
<tr>
<td><code>TSQLite3IndexConstraint</code></td>
<td>Records WHERE clause constraints of the form &quot;column OP expr&quot;</td>
<td>1648</td>
</tr>
<tr>
<td><code>TSQLite3IndexConstraintUsage</code></td>
<td>Define what information is to be passed to xFilter() for a given WHERE clause constraint of the form &quot;column OP expr&quot;</td>
<td>1649</td>
</tr>
<tr>
<td><code>TSQLite3IndexInfo</code></td>
<td>Structure used as part of the virtual table interface to pass information into and receive the reply from the xBestIndex() method of a virtual table module</td>
<td>1649</td>
</tr>
<tr>
<td><code>TSQLite3IndexOrderBy</code></td>
<td>ORDER BY clause, one item per column</td>
<td>1648</td>
</tr>
<tr>
<td><code>TSQLite3Library</code></td>
<td>Wrapper around all SQLite3 library API calls</td>
<td>1659</td>
</tr>
<tr>
<td><code>TSQLite3LibraryDynamic</code></td>
<td>Allow access to an external SQLite3 library engine</td>
<td>1677</td>
</tr>
<tr>
<td><code>TSQLite3MemMethods</code></td>
<td>Defines the interface between SQLite and low-level memory allocation routines</td>
<td>1659</td>
</tr>
<tr>
<td><code>TSQLite3Module</code></td>
<td>Defines a module object used to implement a virtual table.</td>
<td>1652</td>
</tr>
<tr>
<td><code>TSQLite3VTab</code></td>
<td>Virtual Table Instance Object</td>
<td>1651</td>
</tr>
<tr>
<td><code>TSQLite3VTabCursor</code></td>
<td>Virtual Table Cursor Object</td>
<td>1651</td>
</tr>
<tr>
<td><code>TSQLError</code></td>
<td>Wrapper to a SQLite3 request</td>
<td>1678</td>
</tr>
<tr>
<td><code>TSQLStatementCache</code></td>
<td>Used to retrieve a prepared statement</td>
<td>1684</td>
</tr>
<tr>
<td><code>TSQLStatementCached</code></td>
<td>Handle a cache of prepared statements</td>
<td>1684</td>
</tr>
</tbody>
</table>

**TFTSMatchInfo** = packed record

Map the matchinfo function returned BLOB value
- i.e. the default 'pcx' layout, for both FTS3 and FTS4
- see http://www.sqlite.org/fts3.html#matchinfo
- used for the FTS3/FTS4 ranking of results by TSQLRest.FTSMatch method and the internal RANK() function as proposed in http://www.sqlite.org/fts3.html#appendix_a
**TSNSQLite3IndexConstraint = record**

*Records WHERE clause constraints of the form "column OP expr"*
- Where "column" is a column in the virtual table, OP is an operator like "/" or "/<", and EXPR is an arbitrary expression
- So, for example, if the WHERE clause contained a term like this:
  a = 5

Then one of the constraints would be on the "a" column with operator "/=" and an expression of "/5"
- For example, if the WHERE clause contained something like this:
  x BETWEEN 10 AND 100 AND 999>y

The query optimizer might translate this into three separate constraints:
  x >= 10
  x <= 100
  y < 999

**iColumn: integer;**

*Column on left-hand side of constraint*
- The first column of the virtual table is column 0
- The ROWID of the virtual table is column -1
- Hidden columns are counted when determining the column index.

**iTermOffset: integer;**

*Used internally - xBestIndex() should ignore this field*

**op: byte;**

*Constraint operator*
- OP is =, <, <=, >, or >= using one of the SQLITE_INDEX_CONSTRAINT_* values*

**usable: bytebool;**

*True if this constraint is usable*
- The aConstraint[] array contains information about all constraints that apply to the virtual table. But some of the constraints might not be usable because of the way tables are ordered in a join. The xBestIndex method must therefore only consider constraints that have a usable flag which is true, and just ignore contraints with usable set to false

**TSNSQLite3IndexOrderBy = record**

*ORDER BY clause, one item per column*

**desc: bytebool;**

*True for DESC.  False for ASC.*

**iColumn: integer;**

*Column number*
- The first column of the virtual table is column 0
- The ROWID of the virtual table is column -1
- Hidden columns are counted when determining the column index.
**TSQLite3IndexConstraintUsage = record**

Define what information is to be passed to xFilter() for a given WHERE clause constraint of the form "column OP expr"

argvIndex: Integer;

If argvIndex>0 then the right-hand side of the corresponding aConstraint[] is evaluated and becomes the argvIndex-th entry in argv

- Exactly one entry should be set to 1, another to 2, another to 3, and so forth up to as many or as few as the xBestIndex() method wants.
- The EXPR of the corresponding constraints will then be passed in as the argv[] parameters to xFilter()
- For example, if the aConstraint[3].argvIndex is set to 1, then when xFilter() is called, the argv[0] passed to xFilter will have the EXPR value of the aConstraint[3] constraint.

**omit: bytebool;**

If omit is true, then the constraint is assumed to be fully handled by the virtual table and is not checked again by SQLite

- By default, the SQLite core double checks all constraints on each row of the virtual table that it receives. If such a check is redundant, xBestFilter() method can suppress that double-check by setting this field

**TSQLite3IndexInfo = record**

Structure used as part of the virtual table interface to pass information into and receive the reply from the xBestIndex() method of a virtual table module

- Outputs fields will be passed as parameter to the xFilter() method, and will be initialized to zero by SQLite
- For instance, xBestIndex() method fills the idxNum and idxStr fields with information that communicates an indexing strategy to the xFilter method. The information in idxNum and idxStr is arbitrary as far as the SQLite core is concerned. The SQLite core just copies the information through to the xFilter() method. Any desired meaning can be assigned to idxNum and idxStr as long as xBestIndex() and xFilter() agree on what that meaning is. Use the SetInfo() method of this object in order to make a temporary copy of any needed data.

aConstraint: PSQLite3IndexConstraintArray;

*Input: List of WHERE clause constraints of the form "column OP expr"

aConstraintUsage: PSQLite3IndexConstraintUsageArray;

*Output: filled by xBestIndex() method with information about what parameters to pass to xFilter() method*

- has the same number of items than the aConstraint[] array
- should set the aConstraintUsage[].argvIndex to have the corresponding argument in xFilter() argc/argv[] expression list

aOrderBy: PSQLite3IndexOrderByArray;

*Input: List of ORDER BY clause, one per column*
colUsed: UInt64;

*Input: Mask of columns used by statement*  *(since 3.10.0)*
- indicates which fields of the virtual table are actually used by the statement being prepared. If the lowest bit of colUsed is set, that means that the first column is used. The second lowest bit corresponds to the second column. And so forth. If the most significant bit of colUsed is set, that means that one or more columns other than the first 63 columns are used.
- If column usage information is needed by the xFilter method, then the required bits must be encoded into either the idxNum or idxStr output fields

estimatedCost: Double;

*Output: Estimated cost of using this index*
- Should be set to the estimated number of disk access operations required to execute this query against the virtual table
- The SQLite core will often call xBestIndex() multiple times with different constraints, obtain multiple cost estimates, then choose the query plan that gives the lowest estimate

estimatedRows: Int64;

*Output: Estimated number of rows returned*  *(since 3.8.2)*
- may be set to an estimate of the number of rows returned by the proposed query plan. If this value is not explicitly set, the default estimate of 25 rows is used

idxFlags: Integer;

*Output: Mask of SQLITE_INDEX_SCAN_* flags*  *(since 3.9.0)*
- may be set to SQLITE_INDEX_SCAN_UNIQUE to indicate that the virtual table will return only zero or one rows given the input constraints. Additional bits of the idxFlags field might be understood in later versions of SQLite

idxNum: integer;

*Output: Number used to identify the index*

idxStr: PAnsiChar;

*Output: String, possibly obtained from sqlite3.malloc()*
- may contain any variable-length data or class/record content, as necessary

nConstraint: integer;

*Input: Number of entries in aConstraint array*

needToFreeIdxStr: integer;

*Output: Free idxStr using sqlite3.free() if true (=1)*

nOrderBy: integer;

*Input: Number of terms in the aOrderBy array*

orderByConsumed: integer;

*Output: True (=1) if output is already ordered*
- i.e. if the virtual table will output rows in the order specified by the ORDER BY clause
- if False (=0), will indicate to the SQLite core that it will need to do a separate sorting pass over the data after it comes out of the virtual table
**TSQLite3VTab = record**

*Virtual Table Instance Object*
- Every virtual table module implementation uses a subclass of this object to describe a particular instance of the virtual table.
- Each subclass will be tailored to the specific needs of the module implementation. The purpose of this superclass is to define certain fields that are common to all module implementations. This structure therefore contains a pInstance field, which will be used to store a class instance handling the virtual table as a pure Delphi class: the TSQLVirtualTableModule class will use it internally.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nRef: integer;</td>
<td>No longer used</td>
</tr>
<tr>
<td>pInstance: TObject;</td>
<td>This will be used to store a Delphi class instance handling the Virtual Table</td>
</tr>
<tr>
<td>pModule: PSQLite3Module;</td>
<td>The module for this virtual table</td>
</tr>
<tr>
<td>zErrMsg: PUTF8Char;</td>
<td>Error message from sqlite3.mprintf()</td>
</tr>
</tbody>
</table>

- Virtual tables methods can set an error message by assigning a string obtained from sqlite3.mprintf() to zErrMsg.
- The method should take care that any prior string is freed by a call to sqlite3.free() prior to assigning a new string to zErrMsg.
- After the error message is delivered up to the client application, the string will be automatically freed by sqlite3.free() and the zErrMsg field will be zeroed.

**TSQLite3VTabCursor = record**

*Virtual Table Cursor Object*
- Every virtual table module implementation uses a subclass of the following structure to describe cursors that point into the virtual table and are used to loop through the virtual table.
- Cursors are created using the xOpen method of the module and are destroyed by the xClose method. Cursors are used by the xFilter, xNext, xEof, xColumn, and xRowid methods of the module.
- Each module implementation will define the content of a cursor structure to suit its own needs.
- This superclass exists in order to define fields of the cursor that are common to all implementations. This structure therefore contains a pInstance field, which will be used to store a class instance handling the virtual table as a pure Delphi class: the TSQLVirtualTableModule class will use it internally.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pInstance: TObject;</td>
<td>This will be used to store a Delphi class instance handling the cursor</td>
</tr>
<tr>
<td>pVtab: PSQLite3VTab;</td>
<td>Virtual table of this cursor</td>
</tr>
</tbody>
</table>
**TSQLite3Module = record**

*Defines a module object used to implement a virtual table.*
- Think of a module as a class from which one can construct multiple virtual tables having similar properties. For example, one might have a module that provides read-only access to comma-separated-value (CSV) files on disk. That one module can then be used to create several virtual tables where each virtual table refers to a different CSV file.
- The module structure contains methods that are invoked by SQLite to perform various actions on the virtual table such as creating new instances of a virtual table or destroying old ones, reading and writing data, searching for and deleting, updating, or inserting rows.

<table>
<thead>
<tr>
<th>iVersion: integer;</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Defines the particular edition of the module table structure</em></td>
</tr>
</tbody>
</table>
- Currently, handled iVersion is 2, but in future releases of SQLite the module structure definition might be extended with additional methods and in that case the iVersion value will be increased.

<table>
<thead>
<tr>
<th>xBegin: function(var pVTab: TSQLite3VTab): Integer; cdecl;</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Begins a transaction on a virtual table</em></td>
</tr>
</tbody>
</table>
- This method is always followed by one call to either the xCommit or xRollback method.
- Virtual table transactions do not nest, so the xBegin method will not be invoked more than once on a single virtual table without an intervening call to either xCommit or xRollback. For nested transactions, use xSavepoint, xRelease and xRollBackTo methods.
- Multiple calls to other methods can and likely will occur in between the xBegin and the corresponding xCommit or xRollback.

<table>
<thead>
<tr>
<th>xBestIndex: function(var pVTab: TSQLite3VTab; var pInfo: TSQLite3IndexInfo): Integer; cdecl;</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Used to determine the best way to access the virtual table</em></td>
</tr>
</tbody>
</table>
- The pInfo parameter is used for input and output parameters
- The SQLite core calls the xBestIndex() method when it is compiling a query that involves a virtual table. In other words, SQLite calls this method when it is running sqlite3.prepare() or the equivalent.
- By calling this method, the SQLite core is saying to the virtual table that it needs to access some subset of the rows in the virtual table and it wants to know the most efficient way to do that access. The xBestIndex method replies with information that the SQLite core can then use to conduct an efficient search of the virtual table, via the xFilter() method.
- While compiling a single SQL query, the SQLite core might call xBestIndex multiple times with different settings in pInfo. The SQLite core will then select the combination that appears to give the best performance.
- The information in the pInfo structure is ephemeral and may be overwritten or deallocated as soon as the xBestIndex() method returns. If the xBestIndex() method needs to remember any part of the pInfo structure, it should make a copy. Care must be taken to store the copy in a place where it will be deallocated, such as in the idxStr field with needToFreeIdxStr set to 1.
**xClose**: `function(pVtabCursor: PSQLite3VTabCursor): Integer; cdecl;`

*Closes a cursor previously opened by xOpen*
- The SQLite core will always call xClose once for each cursor opened using xOpen.
- This method must release all resources allocated by the corresponding xOpen call.
- The routine will not be called again even if it returns an error. The SQLite core will not use the pVtabCursor after it has been closed.

**xColumn**: `function(var pVtabCursor: TSQLite3VTabCursor; sContext: TSQLite3FunctionContext; N: Integer): Integer; cdecl;`

*The SQLite core invokes this method in order to find the value for the N-th column of the current row*
- N is zero-based so the first column is numbered 0.
- The xColumn method may return its result back to SQLite using one of the standard sqlite3.result_*() functions with the specified sContext.
- If the xColumn method implementation calls none of the sqlite3.result_*() functions, then the value of the column defaults to an SQL NULL.
- The xColumn method must return SQLITE_OK on success.
- To raise an error, the xColumn method should use one of the result_text() methods to set the error message text, then return an appropriate error code.

**xCommit**: `function(var pVTab: TSQLite3VTab): Integer; cdecl;`

*Causes a virtual table transaction to commit*

**xConnect**: `function(DB: TSQLite3DB; pAux: Pointer; argc: Integer; const argv: PPUTF8CharArray; var ppVTab: PSQLite3VTab; var pzErr: PUTF8Char): Integer; cdecl;`

*XConnect is called to establish a new connection to an existing virtual table, whereas xCreate is called to create a new virtual table from scratch*
- It has the same parameters and constructs a new PSQLite3VTab structure.
- xCreate and xConnect methods are only different when the virtual table has some kind of backing store that must be initialized the first time the virtual table is created. The xCreate method creates and initializes the backing store. The xConnect method just connects to an existing backing store.
xCreate: function(DB: TSQLite3DB; pAux: Pointer; argc: Integer; const argv: PPUTF8CharArray; var ppVTab: PSQLite3VTab; var pzErr: PUTF8Char): Integer; cdecl;

Called to create a new instance of a virtual table in response to a CREATE VIRTUAL TABLE statement
- The job of this method is to construct the new virtual table object (an PSQLite3VTab object) and return a pointer to it in ppVTab
- The DB parameter is a pointer to the SQLite database connection that is executing the CREATE VIRTUAL TABLE statement
- The pAux argument is the copy of the client data pointer that was the fourth argument to the sqlite3.create_module_v2() call that registered the virtual table module
- The argv parameter is an array of argc pointers to null terminated strings
- The first string, argv[0], is the name of the module being invoked. The module name is the name provided as the second argument to sqlite3.create_module() and as the argument to the USING clause of the CREATE VIRTUAL TABLE statement that is running.
- The second, argv[1], is the name of the database in which the new virtual table is being created. The database name is "main" for the primary database, or "temp" for TEMP database, or the name given at the end of the ATTACH statement for attached databases.
- The third element of the array, argv[2], is the name of the new virtual table, as specified following the TABLE keyword in the CREATE VIRTUAL TABLE statement
- If present, the fourth and subsequent strings in the argv[] array report the arguments to the module name in the CREATE VIRTUAL TABLE statement
- As part of the task of creating a new PSQLite3VTab structure, this method must invoke sqlite3.declare_vtab() to tell the SQLite core about the columns and datatypes in the virtual table

xDestroy: function(pVTab: PSQLite3VTab): Integer; cdecl;

Releases a connection to a virtual table, just like the xDisconnect method, and it also destroys the underlying table implementation.
- This method undoes the work of xCreate
- The xDisconnect method is called whenever a database connection that uses a virtual table is closed. The xDestroy method is only called when a DROP TABLE statement is executed against the virtual table.

xDISconnect: function(pVTab: PSQLite3VTab): Integer; cdecl;

Releases a connection to a virtual table
- Only the pVTab object is destroyed. The virtual table is not destroyed and any backing store associated with the virtual table persists. This method undoes the work of xConnect.

xEof: function(var pVtabCursor: TSQLite3VTabCursor): Integer; cdecl;

Checks if cursor reached end of rows
- Must return false (zero) if the specified cursor currently points to a valid row of data, or true (non-zero) otherwise
**xFilter**: function(var pVtabCursor: TSQLite3VTabCursor; idxNum: Integer; const idxStr: PAnsiChar; argc: Integer; var argv: TSQLite3ValueArray): Integer; cdecl;

*Begin a search of a virtual table*
- The first argument is a cursor opened by xOpen.
- The next two arguments define a particular search index previously chosen by xBestIndex(). The specific meanings of idxNum and idxStr are unimportant as long as xFilter() and xBestIndex() agree on what that meaning is.
- The xBestIndex() function may have requested the values of certain expressions using the aConstraintUsage[].argvIndex values of its pInfo structure. Those values are passed to xFilter() using the argc and argv parameters.
- If the virtual table contains one or more rows that match the search criteria, then the cursor must be left point at the first row. Subsequent calls to xEof must return false (zero). If there are no rows match, then the cursor must be left in a state that will cause the xEof to return true (non-zero). The SQLite engine will use the xColumn and xRowid methods to access that row content. The xNext method will be used to advance to the next row.
- This method must return SQLITE_OK if successful, or an sqlite error code if an error occurs.

**xFindFunction**: function(var pVTab: TSQLite3VTab; nArg: Integer; const zName: PAnsiChar; var pxFunc: TSQLFunctionFunc; var ppArg: Pointer): Integer; cdecl;

*Called during sqlite3.prepare() to give the virtual table implementation an opportunity to overload SQL functions*
- When a function uses a column from a virtual table as its first argument, this method is called to see if the virtual table would like to overload the function. The first three parameters are inputs: the virtual table, the number of arguments to the function, and the name of the function. If no overloading is desired, this method returns 0. To overload the function, this method writes the new function implementation into pxFunc and writes user data into ppArg and returns 1.
- Note that infix functions (LIKE, GLOB, REGEXP, and MATCH) reverse the order of their arguments. So "like(A,B)" is equivalent to "B like A". For the form "B like A" the B term is considered the first argument to the function. But for "like(A,B)" the A term is considered the first argument.
- The function pointer returned by this routine must be valid for the lifetime of the pVTab object given in the first parameter.

**xNext**: function(var pVtabCursor: TSQLite3VTabCursor): Integer; cdecl;

*Advances a virtual table cursor to the next row of a result set initiated by xFilter*
- If the cursor is already pointing at the last row when this routine is called, then the cursor no longer points to valid data and a subsequent call to the xEof method must return true (non-zero).
- If the cursor is successfully advanced to another row of content, then subsequent calls to xEof must return false (zero).
- This method must return SQLITE_OK if successful, or an sqlite error code if an error occurs.
xOpen: function(var pVtab: TSQLite3VTab; var ppCursor: PSQLite3VTabCursor): Integer; cdecl;

*Creates a new cursor used for accessing (read and/or writing) a virtual table*
- A successful invocation of this method will allocate the memory for the TPSQLite3VTabCursor (or a subclass), initialize the new object, and make ppCursor point to the new object. The successful call then returns SQLITE_OK.
- For every successful call to this method, the SQLite core will later invoke the xClose method to destroy the allocated cursor.
- The xOpen method need not initialize the pVtab field of the ppCursor structure. The SQLite core will take care of that chore automatically.
- A virtual table implementation must be able to support an arbitrary number of simultaneously open cursors.
- When initially opened, the cursor is in an undefined state. The SQLite core will invoke the xFilter method on the cursor prior to any attempt to position or read from the cursor.

xRelease: function(var pVtab: TSQLite3VTab; iSavepoint: integer): Integer; cdecl;
*Merges a transaction into its parent transaction, so that the specified transaction and its parent become the same transaction*
- Causes all savepoints back to and including the most recent savepoint with a matching identifier to be removed from the transaction stack
- Some people view RELEASE as the equivalent of COMMIT for a SAVEPOINT. This is an acceptable point of view as long as one remembers that the changes committed by an inner transaction might later be undone by a rollback in an outer transaction.
- iSavepoint parameter indicates the unique name of the SAVEPOINT

xRename: function(var pVtab: TSQLite3VTab; const zNew: PAnsiChar): Integer; cdecl;
*Provides notification that the virtual table implementation that the virtual table will be given a new name*
- If this method returns SQLITE_OK then SQLite renames the table.
- If this method returns an error code then the renaming is prevented.

xRollback: function(var pVtab: TSQLite3VTab): Integer; cdecl;
*Causes a virtual table transaction to rollback*

xRollbackTo: function(var pVtab: TSQLite3VTab; iSavepoint: integer): Integer; cdecl;
*Reverts the state of the virtual table content back to what it was just after the corresponding SAVEPOINT*
- iSavepoint parameter indicates the unique name of the SAVEPOINT

xRowid: function(var pVtabCursor: TSQLite3VTabCursor; var pRowid: Int64): Integer; cdecl;
*Should fill pRowid with the rowid of row that the virtual table cursor pVtabCursor is currently pointing at*

xSavepoint: function(var pVtab: TSQLite3VTab; iSavepoint: integer): Integer; cdecl;
*Starts a new transaction with the virtual table*
- SAVEPOINTs are a method of creating transactions, similar to BEGIN and COMMIT, except that the SAVEPOINT and RELEASE commands are named and may be nested. See @http://www.sqlite.org/lang_savepoint.html
- iSavepoint parameter indicates the unique name of the SAVEPOINT
xFSYNC: function(var pVTab: TSQLite3VTab): Integer; cdecl;

*Signals the start of a two-phase commit on a virtual table*
- This method is only invoked after call to the xBegin method and prior to an xCommit or xRollback.
- In order to implement two-phase commit, the xSync method on all virtual tables is invoked prior to invoking the xCommit method on any virtual table.
- If any of the xSync methods fail, the entire transaction is rolled back.
xUpdate: function(var pVTab: TSQLite3VTab; nArg: Integer; var ppArg: TSQLite3ValueArray; var pRowid: Int64): Integer; cdecl;

Makes a change to a virtual table content (insert/delete/update)
- The nArg parameter specifies the number of entries in the ppArg[] array
- The value of nArg will be 1 for a pure delete operation or N+2 for an insert or replace or update where N is the number of columns in the table (including any hidden columns)
- The ppArg[0] parameter is the rowid of a row in the virtual table to be deleted. If ppArg[0] is an SQL NULL, then no deletion occurs
- The ppArg[1] parameter is the rowid of a new row to be inserted into the virtual table. If ppArg[1] is an SQL NULL, then the implementation must choose a rowid for the newly inserted row. Subsequent ppArg[] entries contain values of the columns of the virtual table, in the order that the columns were declared. The number of columns will match the table declaration that the xConnect or xCreate method made using the sqlite3.declare_vtab() call. All hidden columns are included.
- When doing an insert without a rowid (nArg>1, ppArg[1] is an SQL NULL), the implementation must set pRowid to the rowid of the newly inserted row; this will become the value returned by the sqlite3.last_insert_rowid() function. Setting this value in all the other cases is a harmless no-op; the SQLite engine ignores the pRowid return value if nArg=1 or ppArg[1] is not an SQL NULL.
- Each call to xUpdate() will fall into one of cases shown below. Note that references to ppArg[i] mean the SQL value held within the ppArg[i] object, not the ppArg[i] object itself:
  nArg = 1
  The single row with rowid equal to ppArg[0] is deleted. No insert occurs.
  nArg > 1
  ppArg[0] = NULL
  A new row is inserted with a rowid ppArg[1] and column values in ppArg[2] and following. If ppArg[1] is an SQL NULL, the a new unique rowid is generated automatically.
  nArg > 1
  ppArg[0] <> NULL
  ppArg[0] = ppArg[1]
  The row with rowid ppArg[0] is updated with new values in ppArg[2] and following parameters.
  nArg > 1
  ppArg[0] <> NULL
  ppArg[0] <> ppArg[1]
  The row with rowid ppArg[0] is updated with rowid ppArg[1] and new values in ppArg[2] and following parameters. This will occur when an SQL statement updates a rowid, as in the statement:
  UPDATE table SET rowid=rowid+1 WHERE ...;
- The xUpdate() method must return SQLITE_OK if and only if it is successful. If a failure occurs, the xUpdate() must return an appropriate error code. On a failure, the pVTab.zErrMsg element may optionally be replaced with a custom error message text.
- If the xUpdate() method violates some constraint of the virtual table (including, but not limited to, attempting to store a value of the wrong datatype, attempting to store a value that is too large or too small, or attempting to change a read-only value) then the xUpdate() must fail with an appropriate error code.
- There might be one or more TSQLite3VTabCursor objects open and in use on the virtual table instance and perhaps even on the row of the virtual table when the xUpdate() method is invoked. The implementation of xUpdate() must be prepared for attempts to delete or modify rows of the table out from other existing cursors. If the virtual table cannot accommodate such changes, the xUpdate() method must return an error code.
TSQLite3MemMethods = record
  Defines the interface between SQLite and low-level memory allocation routines
  - as used by sqlite3.config(SQLITE_CONFIG_MALLOC,pMemMethods);
  pAppData: pointer;
    Argument to xInit() and xShutdown()
  xFree: procedure(ptr: pointer); cdecl;
    Free a prior allocation
  xInit: function(appData: pointer): integer; cdecl;
    Initialize the memory allocator
  xMalloc: function(size: integer): pointer; cdecl;
    Memory allocation function
  xRealloc: function(ptr: pointer; size: integer): pointer; cdecl;
    Resize an allocation
  xRoundup: function(size: integer): integer; cdecl;
    Round up request size to allocation size
  xShutdown: procedure(appData: pointer); cdecl;
    Deinitialize the memory allocator
  xSize: function(ptr: pointer): integer; cdecl;
    Return the size of an allocation
end

TSQLite3Library = class(TObject)
  Wrapper around all SQLite3 library API calls
  - abstract class allowing direct binding of static sqlite3.obj (TSQLite3LibraryStatic) or with an
    external library (TSQLite3LibraryDynamic)
  - a global sqlite3: TSQLite3Library will be defined in this unit, so you should call sqlite3.open() instead of sqlite3_open() for instance
  - if your project refers to SynSQLite3Static unit, it will initialize a TSQLite3LibraryStatic instance
  Used for DI-2.2.1 (page 2548).
aggregate_context: function(Context: TSQLite3FunctionContext; nBytes: integer): pointer; cdecl;

Implementations of aggregate SQL functions use this routine to allocate memory for storing their state.
- The first time the sqlite3.aggregate_context(C,N) routine is called for a particular aggregate function, SQLite allocates N of memory, zeroes out that memory, and returns a pointer to the new memory. On second and subsequent calls to sqlite3.aggregate_context() for the same aggregate function instance, the same buffer is returned. sqlite3.aggregate_context() is normally called once for each invocation of the xStep callback and then one last time when the xFinal callback is invoked. When no rows match an aggregate query, the xStep() callback of the aggregate function implementation is never called and xFinal() is called exactly once. In those cases, sqlite3.aggregate_context() might be called for the first time from within xFinal().
- The amount of space allocated by sqlite3.aggregate_context(C,N) is determined by the N parameter on first successful call. Changing the value of N in subsequent call to sqlite3.aggregate_context() within the same aggregate function instance will not resize the memory allocation.
- SQLite automatically frees the memory allocated by sqlite3.aggregate_context() when the aggregate query concludes.

backup_finish: function(Backup: TSQLite3Backup): integer; cdecl;

Finalize a Backup process on a given database
- When backup_step() has returned SQLITE_DONE, or when the application wishes to abandon the backup operation, the application should destroy the TSQLite3Backup by passing it to backup_finish().
- The backup_finish() interface releases all resources associated with the TSQLite3Backup object. If backup_step() has not yet returned SQLITE_DONE, then any active write-transaction on the destination database is rolled back.
- The TSQLite3Backup object is invalid and may not be used following a call to backup_finish().
- The value returned by backup_finish is SQLITE_OK if no backup_step() errors occurred, regardless or whether or not backup_step() completed. If an out-of-memory condition or IO error occurred during any prior backup_step() call on the same TSQLite3Backup object, then backup_finish() returns the corresponding error code.
- A return of SQLITE_BUSY or SQLITE_LOCKED from backup_step() is not a permanent error and does not affect the return value of backup_finish().
backup_init: function(DestDB: TSQLite3DB; DestDatabaseName: PUTF8Char; SourceDB: TSQLite3DB; SourceDatabaseName: PUTF8Char): TSQLite3Backup; cdecl;

Initialize a backup process of a given SQLite3 database instance
- The DestDB and DestDatabaseName arguments are the database connection associated with the destination database and the database name, respectively. The database name is "main" for the main database, "temp" for the temporary database, or the name specified after the AS keyword in an ATTACH statement for an attached database.
- The SourceDB and SourceDatabaseName arguments identify the database connection and database name of the source database, respectively.
- The source and destination database connections (parameters SourceDB and DestDB) must be different or else function will fail with an error.
- If an error occurs within backup_init(), then nil is returned and an error code and error message are stored in the destination database connection DestDB. The error code and message for the failed call to backup_init() can be retrieved using the errcode() or errmsg() functions.
- A successful call to backup_init() returns a pointer to a TSQLite3Backup object. The TSQLite3Backup object may be used with the backup_step() and backup_finish() functions to perform the specified backup operation.

backup_pagecount: function(Backup: TSQLite3Backup): integer; cdecl;

Returns the total number of pages in the source database file for a given Backup process
- The values returned by this function are only updated by backup_step(). If the source database is modified during a backup operation, then the value is not updated to account for any extra pages that need to be updated or the size of the source database file changing.

backup_remaining: function(Backup: TSQLite3Backup): integer; cdecl;

Returns the number of pages still to be backed up for a given Backup
- The values returned by this function are only updated by backup_step(). If the source database is modified during a backup operation, then the value is not updated to account for any extra pages that need to be updated or the size of the source database file changing.
backup_step: function(Backup: TSQLite3Backup; nPages: integer): integer; cdecl;

Perform a backup step to transfer the data between the two databases
- backup_step() will copy up to nPages pages between the source and destination databases specified by TSQLite3Backup object Backup.
- If nPages is negative, all remaining source pages are copied.
- If backup_step() successfully copies nPages pages and there are still more pages to be copied, then the function returns SQLITE_OK.
- If backup_step() successfully finishes copying all pages from source to destination, then it returns SQLITE_DONE.
- If an error occurs while running backup_step(), an error code is returned.
- As well as SQLITE_OK and SQLITE_DONE, a call to backup_step() may return SQLITE_READONLY, SQLITE_NOMEM, SQLITE_BUSY, SQLITE_LOCKED, or an SQLITE_IOERR_XXX extended error code. The function might return SQLITE_READONLY if the destination database was opened read-only, or is using WAL journaling and the destination and source page sizes differ, or the destination database is an in-memory database and the destination and source page sizes differ. SQLITE_BUSY indicates that the file-system lock did not succeed: in this case the call to backup_step() can be retried later. If the source database connection is being used to write to the source database when backup_step() is called, then SQLITE_LOCKED is returned immediately. Again, in this case the call to backup_step() can be retried later on. If SQLITE_IOERR_XXX, SQLITE_NOMEM, or SQLITE_READONLY is returned, then there is no point in retrying the call to backup_step(). These errors are considered fatal. The application must accept that the backup operation has failed and pass the backup operation handle to the backup_finish() to release associated resources.
- The first call to sqlite3_backup_step() obtains an exclusive lock on the destination file. The exclusive lock is not released until either backup_finish() is called or the backup operation is complete and backup_step() returns SQLITE_DONE. Every call to backup_step() obtains a shared lock on the source database that lasts for the duration of the backup_step() call.
- Because the source database is not locked between calls to backup_step(), the source database may be modified mid-way through the backup process. If the source database is modified by an external process or via a database connection other than the one being used by the backup operation, then the backup will be automatically restarted by the next call to backup_step(). If the source database is modified by the using the same database connection as is used by the backup operation (which is the case in the SynSQLite3 and mORMotSQLite3 units), then the backup database is automatically updated at the same time, so you won't loose any data.

bind_blob: function(S: TSQLite3Statement; Param: integer; Buf: pointer; Buf_bytes: integer; DestroyPtr: TSQLDestroyPtr=SQLITE_TRANSIENT): integer; cdecl;

Note that the official SQLite3 documentation could lead into misunderstanding: Text_bytes must EXCLUDE the null terminator, otherwise a #0 is appended to all column values Bind a Blob Value to a parameter of a prepared statement
- return SQLITE_OK on success or an error code - see SQLITE_* and sqlite3.errmsg()
- S is a statement prepared by a previous call to sqlite3.prepare_v2()
- Param is the index of the SQL parameter to be set (leftmost=1)
- Buf must point to a memory buffer of Buf_bytes bytes
- Buf_bytes contains the number of bytes in Buf
- set DestroyPtr as SQLITE_STATIC (nil) for static binding
- set DestroyPtr to SQLITE_TRANSIENT (-1) for SQLite to make its own private copy of the data (this is the preferred way in our Framework)
- set DestroyPtr to @sqlite3InternalFree if Value must be released via Freemem()
bind_double: function(S: TSQLite3Statement; Param: integer; Value: double): integer; cdecl;
   * Bind a floating point Value to a parameter of a prepared statement*
   - return SQLITE_OK on success or an error code - see SQLITE_* and sqlite3.errmsg()
   - S is a statement prepared by a previous call to sqlite3.prepare_v2()
   - Param is the index of the SQL parameter to be set (leftmost=1)
   - Value is the floating point number to bind

bind_int: function(S: TSQLite3Statement; Param: integer; Value: integer): integer; cdecl;
   * Bind a 32 bits Integer Value to a parameter of a prepared statement*
   - return SQLITE_OK on success or an error code - see SQLITE_* and sqlite3.errmsg()
   - S is a statement prepared by a previous call to sqlite3.prepare_v2()
   - Param is the index of the SQL parameter to be set (leftmost=1)
   - Value is the 32 bits Integer to bind

bind_int64: function(S: TSQLite3Statement; Param: integer; Value: Int64): integer; cdecl;
   * Bind a 64 bits Integer Value to a parameter of a prepared statement*
   - return SQLITE_OK on success or an error code - see SQLITE_* and sqlite3.errmsg()
   - S is a statement prepared by a previous call to sqlite3.prepare_v2()
   - Param is the index of the SQL parameter to be set (leftmost=1)
   - Value is the 64 bits Integer to bind

bind_null: function(S: TSQLite3Statement; Param: integer): integer; cdecl;
   * Bind a NULL Value to a parameter of a prepared statement*
   - return SQLITE_OK on success or an error code - see SQLITE_* and sqlite3.errmsg()
   - S is a statement prepared by a previous call to sqlite3.prepare_v2()
   - Param is the index of the SQL parameter to be set (leftmost=1)

bind_parameter_count: function(S: TSQLite3Statement): integer; cdecl;
   * Number Of SQL Parameters for a prepared statement*
   - returns the index of the largest (rightmost) parameter. For all forms except ?NNN, this will
     correspond to the number of unique parameters.
   - If parameters of the ?NNN type are used, there may be gaps in the list.

bind_text: function(S: TSQLite3Statement; Param: integer; Text: PUTF8Char; Text_bytes: integer=-1; DestroyPtr: TSQLDestroyPtr=SQLITE_TRANSIENT): integer; cdecl;
   * Bind a Text Value to a parameter of a prepared statement*
   - return SQLITE_OK on success or an error code - see SQLITE_* and sqlite3.errmsg()
   - S is a statement prepared by a previous call to sqlite3.prepare_v2()
   - Param is the index of the SQL parameter to be set. The leftmost SQL parameter has an index of
     1.
   - Text must contains an UTF8-encoded null-terminated string query
   - Text_bytes contains -1 (to stop at the null char) or the number of chars in the input string,
     excluding the null terminator
   - set DestroyPtr as SQLITE_STATIC (nil) for static binding
   - set DestroyPtr to SQLITE_TRANSIENT (-1) for SQLite to make its own private copy of the data
     (this is the preferred way in our Framework)
   - set DestroyPtr to @$sqlite3InternalFree if Value must be released via Freemem()
bind_zeroblob: function(S: TSQLite3Statement; Param: integer; Size: integer): integer; cdecl;

Bind a ZeroBlob buffer to a parameter
- uses a fixed amount of memory (just an integer to hold its size) while it is being processed.
Zeroblobs are intended to serve as placeholders for BLOBs whose content is later written using incrementally BLOB I/O routines.
- a negative value for the Size parameter results in a zero-length BLOB
- the leftmost SQL parameter has an index of 1, but ?NNN may override it

blob_bytes: function(Blob: TSQLite3Blob): integer; cdecl;

Return The Size Of An Open BLOB

blob_close: function(Blob: TSQLite3Blob): integer; cdecl;

Close A BLOB Handle

blob_open: function(DB: TSQLite3DB; DBName, TableName, ColumnName: PUTF8Char; RowID: Int64; Flags: Integer; var Blob: TSQLite3Blob): integer; cdecl;

Open a BLOB For Incremental I/O
- returns a BLOB handle for row RowID, column ColumnName, table TableName in database
DBName; in other words, the same BLOB that would be selected by:
SELECT ColumnName FROM DBName.TableName WHERE rowid = RowID;

blob_read: function(Blob: TSQLite3Blob; const Data; Count, Offset: integer): integer; cdecl;

Read Data From a BLOB Incrementally

blob_reopen: function(Blob: TSQLite3Blob; RowID: Int64): integer; cdecl;

Move a BLOB Handle to a New Row
- will point to a different row of the same database table
- this is faster than closing the existing handle and opening a new one

blob_write: function(Blob: TSQLite3Blob; const Data; Count, Offset: Integer): integer; cdecl;

Write Data To a BLOB Incrementally

busy_handler: function(DB: TSQLite3DB; CallbackPtr: TSQLiteBusyHandler; user: pointer): integer; cdecl;

Register A Callback To Handle SQLITE_BUSY Errors
- This routine sets a callback function that might be invoked whenever an attempt is made to open a database table that another thread or process has locked.
- If the busy callback is nil, then SQLITE_BUSY or SQLITE_IOERR_BLOCKED is returned immediately upon encountering the lock. If the busy callback is not nil, then the callback might be invoked with two arguments.
- The default busy callback is nil.
**busy_timeout**: function(DB: TSQLite3DB; Milliseconds: integer): integer; cdecl;

*Set A Busy Timeout*
- This routine sets a busy handler that sleeps for a specified amount of time when a table is locked. The handler will sleep multiple times until at least "ms" milliseconds of sleeping have accumulated. After at least "ms" milliseconds of sleeping, the handler returns 0 which causes sqlite3.step() to return SQLITE_BUSY or SQLITE_IOERR_BLOCKED.
- Calling this routine with an argument less than or equal to zero turns off all busy handlers.
- There can only be a single busy handler for a particular database connection any given moment. If another busy handler was defined (using sqlite3.busy_handler()) prior to calling this routine, that other busy handler is cleared.

**changes**: function(DB: TSQLite3DB): Integer; cdecl;

*Count The Number Of Rows Modified*
- This function returns the number of database rows that were changed or inserted or deleted by the most recently completed SQL statement on the database connection specified by the first parameter. Only changes that are directly specified by the INSERT, UPDATE, or DELETE statement are counted. Auxiliary changes caused by triggers or foreign key actions are not counted. Use the sqlite3.total_changes() function to find the total number of changes including changes caused by triggers and foreign key actions.
- If a separate thread makes changes on the same database connection while sqlite3.changes() is running then the value returned is unpredictable and not meaningful.

**clear_bindings**: function(S: TSQLite3Statement): integer; cdecl;

*Reset All Bindings On A Prepared Statement*

**close**: function(DB: TSQLite3DB): integer; cdecl;

*Destructor for the sqlite3 object, which handle is DB*
- Applications should finalize all prepared statements and close all BLOB handles associated with the sqlite3 object prior to attempting to close the object (sqlite3.next_stmt() interface can be used for this task)
- if invoked while a transaction is open, the transaction is automatically rolled back
- SynSQLite3Static will use its own internal function for handling properly its own encryption format

**column_blob**: function(S: TSQLite3Statement; Col: integer): PAnsiChar; cdecl;

*Converts the Col column in the current row of prepared statement S into a BLOB and then returns a pointer to the converted value*
- NULL is converted into nil
- INTEGER or FLOAT are converted into ASCII rendering of the numerical value
- TEXT and BLOB are returned directly

**column_bytes**: function(S: TSQLite3Statement; Col: integer): integer; cdecl;

*Number of bytes for a BLOB or UTF-8 string result*
- S is the SQL statement, after sqlite3.step(S) returned SQLITE_ROW
- Col is the column number, indexed from 0 to sqlite3.column_count(S)-1
- an implicit conversion into UTF-8 text is made for a numeric value or UTF-16 column: you must call sqlite3.column_text() or sqlite3.column_blob() before calling sqlite3.column_bytes() to perform the conversion itself

**column_count**: function(S: TSQLite3Statement): integer; cdecl;

*Get the number of columns in the result set for the statement*
column_decltype: function(S: TSQLite3Statement; Col: integer): PAnsiChar; cdecl;
  Returns a zero-terminated UTF-8 string containing the declared datatype of a result column.

column_double: function(S: TSQLite3Statement; Col: integer): double; cdecl;
  Converts the Col column in the current row prepared statement S into a floating point value and returns a copy of that value.
  - NULL is converted into 0.0
  - INTEGER is converted into corresponding floating point value
  - TEXT or BLOB is converted from all correct ASCII numbers with 0.0 as default.

column_int: function(S: TSQLite3Statement; Col: integer): integer; cdecl;
  Converts the Col column in the current row prepared statement S into a 32 bit integer value and returns a copy of that value.
  - NULL is converted into 0
  - FLOAT is truncated into corresponding integer value
  - TEXT or BLOB is converted from all correct ASCII numbers with 0 as default.

column_int64: function(S: TSQLite3Statement; Col: integer): int64; cdecl;
  Converts the Col column in the current row prepared statement S into a 64 bit integer value and returns a copy of that value.
  - NULL is converted into 0
  - FLOAT is truncated into corresponding integer value
  - TEXT or BLOB is converted from all correct ASCII numbers with 0 as default.

column_name: function(S: TSQLite3Statement; Col: integer): PUTF8Char; cdecl;
  Returns the name of a result column as a zero-terminated UTF-8 string.

column_text: function(S: TSQLite3Statement; Col: integer): PUTF8Char; cdecl;
  Converts the Col column in the current row prepared statement S into a zero-terminated UTF-8 string and returns a pointer to that string.
  - NULL is converted into nil
  - INTEGER or FLOAT are converted into ASCII rendering of the numerical value
  - TEXT is returned directly (with UTF-16 -> UTF-8 encoding if necessary)
  - BLOB add a zero terminator if needed.

column_text16: function(S: TSQLite3Statement; Col: integer): PWideChar; cdecl;
  Converts the Col column in the current row prepared statement S into a zero-terminated UTF-16 string and returns a pointer to that string.
  - NULL is converted into nil
  - INTEGER or FLOAT are converted into ASCII rendering of the numerical value
  - TEXT is returned directly (with UTF-8 -> UTF-16 encoding if necessary)
  - BLOB add a zero terminator if needed.

column_type: function(S: TSQLite3Statement; Col: integer): integer; cdecl;
  Datatype code for the initial data type of a result column.
  - returned value is one of SQLITE_INTEGER, SQLITE_FLOAT, SQLITE_TEXT, SQLITE_BLOB or SQLITE_NULL
  - S is the SQL statement, after sqlite3.step(S) returned SQLITE_ROW
  - Col is the column number, indexed from 0 to sqlite3.column_count(S)-1
  - must be called before any sqlite3.column_*() statement, which may result in an implicit type conversion: in this case, value is undefined.
column_value: function(S: TSQLite3Statement; Col: integer): TSQLite3Value; cdecl;
   Get the value handle of the Col column in the current row of prepared statement S
   - this handle represent a sqlite3.value object
   - this handle can then be accessed with any sqlite3.value_*() function below

commit_hook: function(DB: TSQLite3DB; xCallback: TSQLCommitCallback; pArg: Pointer): Pointer; cdecl;
   Register Commit Notification Callbacks
   - The sqlite3.commit_hook() interface registers a callback function to be invoked whenever a
     transaction is committed.
   - Any callback set by a previous call to sqlite3.commit_hook() for the same database connection
     is overridden.
   - Registering a nil function disables the Commit callback.
   - The sqlite3.commit_hook(DB,C,P) function returns the P argument from the previous call of the
     same function on the same database connection DB, or nil for the first call for each function on
     DB.

config: function(operation: integer): integer; cdecl varargs;
   Used to make global configuration changes to current database

context_db_handle: function(Context: TSQLite3FunctionContext): TSQLite3DB; cdecl;
   Returns a copy of the pointer to the database connection (the 1st parameter) of the
   sqlite3.create_function() routine that originally registered the application defined function

create_collation: function(DB: TSQLite3DB; CollationName: PUTF8Char;
   StringEncoding: integer; CollateParam: pointer; cmp: TSQLCollateFunc): integer; cdecl;
   Define New Collating Sequences
   - add new collation sequences to the database connection specified
   - collation name is to be used in CREATE TABLE t1 (a COLLATE CollationName); or in SELECT *
     FROM t1 ORDER BY c COLLATE CollationName;
   - StringEncoding is either SQLITE_UTF8 either SQLITE_UTF16
   - TSQLDataBase.Create add WIN32CASE, WIN32NOCASE and ISO8601 collations
create_function: function(DB: TSQLite3DB; FunctionName: PUTF8Char; nArg, eTextRep: integer; pApp: pointer; xFunc, xStep: TSQLFunctionFunc; xFinal: TSQLFunctionFinal): Integer; cdecl;

Add SQL functions or aggregates or to redefine the behavior of existing SQL functions or aggregates

- The first parameter is the database connection to which the SQL function is to be added. If an application uses more than one database connection then application-defined SQL functions must be added to each database connection separately.
- The second parameter is the name of the SQL function to be created or redefined. The length of the name is limited to 255 bytes in a UTF-8 representation, exclusive of the zero-terminator. Note that the name length limit is in UTF-8 bytes, not characters nor UTF-16 bytes. Any attempt to create a function with a longer name will result in SQLITE_MISUSE being returned.
- The third parameter (nArg) is the number of arguments that the SQL function or aggregate takes. If this parameter is -1, then the SQL function or aggregate may take any number of arguments between 0 and the SQLITE_LIMIT_FUNCTION_ARG current limit. If the third parameter is less than -1 or greater than 127 then the behavior is undefined.
- The fourth parameter, eTextRep, specifies what text encoding this SQL function prefers for its parameters. Every SQL function implementation must be able to work with UTF-8, UTF-16le, or UTF-16be. But some implementations may be more efficient with one encoding than another. When multiple implementations of the same function are available, SQLite will pick the one that involves the least amount of data conversion. If there is only a single implementation which does not care what text encoding is used, then the fourth argument should be SQLITE_ANY.
- The fifth parameter, pApp, is an arbitrary pointer. The implementation of the function can gain access to this pointer using sqlite3.user_data().
- The seventh, eighth and ninth parameters, xFunc, xStep and xFinal, are pointers to C-language functions that implement the SQL function or aggregate. A scalar SQL function requires an implementation of the xFunc callback only; nil pointers must be passed as the xStep and xFinal parameters. An aggregate SQL function requires an implementation of xStep and xFinal and nil pointer must be passed for xFunc. To delete an existing SQL function or aggregate, pass nil pointers for all three function callbacks.
- It is permitted to register multiple implementations of the same functions with the same name but with either differing numbers of arguments or differing preferred text encodings. SQLite will use the implementation that most closely matches the way in which the SQL function is used.

create_function_v2: function(DB: TSQLite3DB; FunctionName: PUTF8Char; nArg, eTextRep: integer; pApp: pointer; xFunc, xStep: TSQLFunctionFunc; xFinal: TSQLFunctionFinal; xDestroy: TSQLDestroyPtr): Integer; cdecl;

Add SQL functions or aggregates or to redefine the behavior of existing SQL functions or aggregates, including destruction

- if the additional xDestroy parameter is not nil, then it is invoked when the function is deleted, either by being overloaded or when the database connection closes.
- When the destructure callback of the tenth parameter is invoked, it passed a single argument which is a copy of the pointer which was the fifth parameter to sqlite3.create_function_v2().
- this function is not available in older revisions - e.g. 3.6.*
**create_module_v2**: function(DB: TSQLite3DB; const zName: PAnsiChar; var p: TSQLite3Module; pClientData: Pointer; xDestroy: TSQLDestroyPtr): Integer; cdecl;

*Used to register a new virtual table module name*
- The module name is registered on the database connection specified by the first DB parameter.
- The name of the module is given by the second parameter.
- The third parameter is a pointer to the implementation of the virtual table module.
- The fourth parameter is an arbitrary client data pointer that is passed through into the xCreate and xConnect methods of the virtual table module when a new virtual table is being created or reinitialized.
- The fifth parameter can be used to specify a custom destructor for the pClientData buffer. SQLite will invoke the destructor function (if it is not nil) when SQLite no longer needs the pClientData pointer. The destructor will also be invoked if call to sqlite3.create_module_v2() fails.

**create_window_function**: function(DB: TSQLite3DB; FunctionName: PUTF8Char; nArg, eTextRep: integer; pApp: pointer; xStep: TSQLFunctionFunc; xFinal, xValue: TSQLFunctionFinal; xInverse: TSQLFunctionFunc; xDestroy: TSQLDestroyPtr): Integer; cdecl;

*Add SQL functions or aggregates or to redefine the behavior of existing SQL functions or aggregates, including extra callback functions needed by aggregate window functions*
- see https://www.sqlite.org/windowfunctions.html#aggregate_window_functions
- sixth, seventh, eighth and ninth parameters (xStep, xFinal, xValue and xInverse) passed to this function are pointers to callbacks that implement the new aggregate window function. xStep and xFinal must both be non-nil. xValue and xInverse may either both be nil, in which case a regular aggregate function is created, or must both be non-nil, in which case the new function may be used as either an aggregate or aggregate window function
- this function is not available in older revisions, i.e. before 3.25.2

**db_config**: function(DestDB: TSQLite3DB; operation: integer): integer; cdecl

*varargs*

*Used to make global configuration changes to current database connection*

**declare_vtab**: function(DB: TSQLite3DB; const zSQL: PAnsiChar): Integer; cdecl;

*Declare the Schema of a virtual table*
- The xCreate() and xConnect() methods of a virtual table module call this interface to declare the format (the names and datatypes of the columns) of the virtual tables they implement. The string can be deallocated and/or reused as soon as the sqlite3.declare_vtab() routine returns.
- If a column datatype contains the special keyword "HIDDEN" (in any combination of upper and lower case letters) then that keyword it is omitted from the column datatype name and the column is marked as a hidden column internally. A hidden column differs from a normal column in three respects: 1. Hidden columns are not listed in the dataset returned by "PRAGMA table_info", 2. Hidden columns are not included in the expansion of a "*" expression in the result set of a SELECT, and 3. Hidden columns are not included in the implicit column-list used by an INSERT statement that lacks an explicit column-list.
deserialize: function(DB: TSQLite3DB; Schema: PUTF8Char; Data: pointer; DBSize, BufSize: Int64; Flags: integer): pointer; cdecl;

Deserialze a database
- causes the database connection DB to disconnect from database Schema and then reopen Schema as an in-memory database based on the serialization contained in Data; the serialized database Data is DBSize bytes in size
- BufSize is the size of the buffer Data, which might be larger than DBSize

errmsg: function(DB: TSQLite3DB): PUTF8Char; cdecl;

Returns English-language text that describes an error, using UTF-8 encoding (which, with English text, is the same as Ansi).
- Memory to hold the error message string is managed internally. The application does not need to worry about freeing the result. However, the error string might be overwritten or deallocated by subsequent calls to other SQLite interface functions.

extended_errcode: function(DB: TSQLite3DB): integer; cdecl;

Returns the numeric result code or extended result code for the most recent failed sqlite3 API call associated with a database connection

finalize: function(S: TSQLite3Statement): integer; cdecl;

Delete a previously prepared statement
- return SQLITE_OK on success or an error code - see SQLITE_* and sqlite3.errmsg()
- this routine can be called at any point during the execution of the prepared statement. If the virtual machine has not completed execution when this routine is called, that is like encountering an error or an interrupt. Incomplete updates may be rolled back and transactions canceled, depending on the circumstances, and the error code returned will be SQLITE_ABORT

free_: procedure(p: Pointer); cdecl;

Releases memory previously returned by sqlite3.malloc() or sqlite3.realloc()
- should call native free() function, i.e. FreeMem() in this unit
- renamed free_ in order not to override TObject.Free method

initialize: function: integer; cdecl;

Initialize the SQLite3 database code
- automatically called by the initialization block of this unit
- so sqlite3.c is compiled with SQLITE_OMIT_AUTOINIT defined

key: function(DB: TSQLite3DB; key: pointer; keyLen: Integer): integer; cdecl;

Specify the encryption key on a newly opened database connection
- Assigned(key)=false if encryption is not available for this .dll
- SynSQLite3Static will use its own internal encryption format
- key/keylen may be a JSON-serialized TSynSignerParams object, or will use AES-OFB-128 after SHAKE_128 with rounds=1000 and a fixed salt on plain password text

last_insert_rowid: function(DB: TSQLite3DB): Int64; cdecl;

Returns the rowid of the most recent successful INSERT into the database

libversion: function: PUTF8Char; cdecl;

Return the version of the SQLite database engine, in ascii format
- currently returns ‘3.33.0’, when used with our SynSQLite3Static unit
- if an external SQLite3 library is used, version may vary
- you may use the VersionText property (or Version for full details) instead
limit: function(DB: TSQLite3DB; id,newValue: integer): integer; cdecl;

Allows the size of various constructs to be limited on a connection by connection basis
- The first parameter is the database connection whose limit is to be set or queried
- The second parameter is one of the limit categories that define a class of constructs to be size limited - see TSQLLimitCategory enumerate
- The third parameter is the new limit for that construct. If the new limit is a negative number, the limit is unchanged.
- Regardless of whether or not the limit was changed, the sqlite3.limit() interface returns the prior value of the limit. Hence, to find the current value of a limit without changing it, simply invoke this interface with the third parameter set to -1.

malloc: function(n: Integer): Pointer; cdecl;

Returns a pointer to a block of memory at least N bytes in length
- should call native malloc() function, i.e. GetMem() in this unit

memory_highwater: function(resetFlag: Integer): Int64; cdecl;

Returns the maximum value of sqlite3.memory_used() since the high-water mark was last reset

memory_used: function: Int64; cdecl;

Returns the number of bytes of memory currently outstanding (malloced but not freed)

next_stmt: function(DB: TSQLite3DB; S: TSQLite3Statement): TSQLite3Statement; cdecl;

Find the next prepared statement
- this interface returns a handle to the next prepared statement after S, associated with the database connection DB.
- if S is 0 then this interface returns a pointer to the first prepared statement associated with the database connection DB.
- if no prepared statement satisfies the conditions of this routine, it returns 0

open: function(filename: PUTF8Char; var DB: TSQLite3DB): integer; cdecl;

Open a SQLite3 database filename, creating a DB handle
- filename must be UTF-8 encoded (filenames containing international characters must be converted to UTF-8 prior to passing them)
- allocate a sqlite3 object, and return its handle in DB
- return SQLITE_OK on success
- an error code (see SQLITE_* const) is returned otherwise - sqlite3.errmsg() can be used to obtain an English language description of the error
- Whatever or not an error occurs when it is opened, resources associated with the database connection handle should be released by passing it to sqlite3.close() when it is no longer required
**open_v2: function**

- **filename**: PUTF8Char
- **var DB**: TSQLite3DB
- **flags**: integer
- **zVfszVfs**: PUTF8Char

**Open a SQLite3 database filename, creating a DB handle**
- sqlite3.open_v2() interface works like sqlite3.open() except that it accepts two additional parameters for additional control over the new database connection.
- flags parameter to sqlite3.open_v2() can take one of
  - SQLITE_OPEN_READONLY,
  - SQLITE_OPEN_READWRITE,
  - or (SQLITE_OPEN_READONLY or SQLITE_OPEN_READWRITE or SQLITE_OPEN_CREATE) values,
  - optionally combined with the
    - SQLITE_OPEN_NOMUTEX, SQLITE.OPEN_FULLMUTEX,
    - SQLITE.OPEN_SHAREDCA ache, SQLITE.OPEN_PRIVATECACHE, and/or SQLITE.OPEN_URI flags
- If the flags parameter is not one of the combinations shown above optionally combined with other SQLite.OPEN_* bits then the behavior is undefined.
- The fourth parameter is the name of the sqlite3_vfs object that defines the operating system interface that the new database connection should use. If the fourth parameter is a nil pointer then the default sqlite3_vfs object is used.

**prepare_v2: function**

- **DB**: TSQLite3DB
- **SQL**: PUTF8Char
- **SQL_bytes**: integer
- **var S**: TSQLite3Statement
- **var SQLtail**: PUTF8Char

**Compile a SQL query into byte-code**
- SQL must contains an UTF8-encoded null-terminated string query
- SQL_bytes contains -1 (to stop at the null char) or the number of bytes in the input string, including the null terminator
- return SQLITE_OK on success or an error code - see SQLITE_* and sqlite3.errmsg() 
- S will contain an handle of the resulting statement (an opaque sqlite3.stmt object) on success, or will 0 on error - the calling procedure is responsible for deleting the compiled SQL statement using sqlite3.finalize() after it has finished with it
- in this "v2" interface, the prepared statement that is returned contains a copy of the original SQL text
- this routine only compiles the first statement in SQL, so SQLtail is left pointing to what remains uncompiled

**realloc: function**

- **pOld**: Pointer
- **n**: Integer

**Attempts to resize a prior memory allocation**
- should call native realloc() function, i.e. ReallocMem() in this unit

**rekey: function**

- **DB**: TSQLite3DB
- **key**: pointer
- **keyLen**: Integer

**Change the encryption key on a database connection that is already opened**
- can also decrypt a previously encrypted database (so that it is accessible from any version of SQLite) by specifying a nil key
- Assigned(rekey)=false if encryption is not available, i.e. if NOSQLITE3STATIC is defined
- also see ChangeSQLEncryptTablePassWord() procedure

**reset: function**

- **S**: TSQLite3Statement

**Reset a prepared statement object back to its initial state, ready to be re-Prepared**
- if the most recent call to sqlite3.step(S) returned SQLITE_ROW or SQLITE_DONE, or if sqlite3.step(S) has never before been called with S, then sqlite3.reset(S) returns SQLITE_OK.
- return an appropriate error code if the most recent call to sqlite3.step(S) failed
- any SQL statement variables that had values bound to them using the sqlite3.bind_*() API retain their values. Use sqlite3.clear_bindings() to reset the bindings.
result_blob: procedure(Context: TSQLite3FunctionContext; Value: Pointer; Value_bytes: Integer=0; DestroyPtr: TSQLDestroyPtr=SQLITE_TRANSIENT); cdecl;

Sets the result from an application-defined function to be the BLOB
- content is pointed to by the Value and which is Value_bytes bytes long
- set DestroyPtr as SQLITE_STATIC (nil) for static binding
- set DestroyPtr to SQLITE_TRANSIENT (-1) for SQLite to make its own private copy of the data (this is the preferred way in our Framework)
- set DestroyPtr to @sqlite3InternalFree if Value must be released via Freemem() or to @sqlite3InternalFreeObject if Value must be released via a Free method

result_double: procedure(Context: TSQLite3FunctionContext; Value: double); cdecl;

Sets the result from an application-defined function to be a floating point value specified by its 2nd argument

result_error: procedure(Context: TSQLite3FunctionContext; Msg: PUTF8Char; MsgLen: integer=-1); cdecl;

Cause the implemented SQL function to throw an exception
- SQLite interprets the error message string from sqlite3.result_error() as UTF-8
- if Msglen is negative, Msg must be #0 ended, or MsgLen must tell the number of characters in the Msg UTF-8 buffer

result_int64: procedure(Context: TSQLite3FunctionContext; Value: Int64); cdecl;

Sets the return value of the application-defined function to be the 64-bit signed integer value given in the 2nd argument

result_null: procedure(Context: TSQLite3FunctionContext); cdecl;

Sets the return value of the application-defined function to be NULL

result_text: procedure(Context: TSQLite3FunctionContext; Value: PUTF8Char; Value_bytes: Integer=-1; DestroyPtr: TSQLDestroyPtr=SQLITE_TRANSIENT); cdecl;

Sets the return value of the application-defined function to be a text string which is represented as UTF-8
- if Value_bytes is negative, then SQLite takes result text from the Value parameter through the first zero character
- if Value_bytes is non-negative, then as many bytes (NOT characters: this parameter must include the #0 terminator) of the text pointed to by the Value parameter are taken as the application-defined function result
- set DestroyPtr as SQLITE_STATIC (nil) for static binding
- set DestroyPtr to SQLITE_TRANSIENT (-1) for SQLite to make its own private copy of the data (this is the preferred way in our Framework)
- set DestroyPtr to @sqlite3InternalFree if Value must be released via Freemem() or to @sqlite3InternalFreeObject if Value must be released via a Free method

result_value: procedure(Context: TSQLite3FunctionContext; Value: TSQLite3Value); cdecl;

Sets the result of the application-defined function to be a copy the unprotected sqlite3.value object specified by the 2nd parameter
- The sqlite3.result_value() interface makes a copy of the sqlite3.value so that the sqlite3.value specified in the parameter may change or be deallocated after sqlite3.result_value() returns without harm
rollback_hook: function(DB: TSQLite3DB; xCallback: TSQLCommitCallback; pArg: Pointer): Pointer; cdecl;

Register Rollback Notification Callbacks
- The sqlite3.rollback_hook() interface registers a callback function to be invoked whenever a transaction is rolled back.
- Any callback set by a previous call to sqlite3.rollback_hook() for the same database connection is overridden.
- Registering a nil function disables the Rollback callback.
- The sqlite3.rollback_hook(D,C,P) function returns the P argument from the previous call of the same function on the same database connection D, or nil for the first call for each function on D.

serialize: function(DB: TSQLite3DB; Schema: PUTF8Char; Size: PInt64; Flags: integer): pointer; cdecl;

Serialize a database
- returns a pointer to memory that is a serialization of the Schema database on database connection DB
- if Size is not nil, then the size of the database in bytes is written into Size
- for an ordinary on-disk database file, the serialization is just a copy of the disk file; for an in-memory database or a "TEMP" database, the serialization is the same sequence of bytes which would be written to disk if that database where backed up to disk
- caller is responsible for freeing the returned value (using free_) to avoid a memory leak

set_authorizer: function(DB: TSQLite3DB; xAuth: TSQLAuthorizerCallback; pUserData: Pointer): Integer; cdecl;

Registers an authorizer callback to a specified DB connection
- Only a single authorizer can be in place on a database connection at a time
- Each call to sqlite3.set_authorizer overrides the previous call
- Disable the authorizer by installing a nil callback
- The authorizer is disabled by default

shutdown: function: integer; cdecl;

Shutdown the SQLite3 database core
- automatically called by the finalization block of this unit

soft_heap_limit64: function(N: Int64): Int64; cdecl;

Sets and/or queries the soft limit on the amount of heap memory that may be allocated by SQLite
- SQLite strives to keep heap memory utilization below the soft heap limit by reducing the number of pages held in the page cache as heap memory usages approaches the limit. The soft heap limit is "soft" because even though SQLite strives to stay below the limit, it will exceed the limit rather than generate an SQLITE_NOMEM error. In other words, the soft heap limit is advisory only
- The return value from soft_heap_limit64() is the size of the soft heap limit prior to the call, or negative in the case of an error. If the argument N is negative then no change is made to the soft heap limit. Hence, the current size of the soft heap limit can be determined by invoking soft_heap_limit64() with a negative argument
- This function is useful when you have many SQLite databases open at the same time, as the cache-size setting is per-database (connection), while this limit is global for the process, so this allows to limit the total cache size
**step**: function(S: TSQLite3Statement): integer; cdecl;

*Evaluate An SQL Statement, returning a result status:*

- SQLITE_BUSY means that the database engine was unable to acquire the database locks it needs to do its job. If the statement is a COMMIT or occurs outside of an explicit transaction, then you can retry the statement. If the statement is not a COMMIT and occurs within an explicit transaction then you should rollback the transaction before continuing.

- SQLITE_DONE means that the statement has finished executing successfully. sqlite3.step() should not be called again on this virtual machine without first calling sqlite3.reset() to reset the virtual machine state back.

- SQLITE_ROW is returned each time a new row of data is ready for processing by the caller. The values may be accessed using the column access functions below. sqlite3.step() has to be called again to retrieve the next row of data.

- SQLITE_MISUSE means that the this routine was called inappropriately. Perhaps it was called on a prepared statement that has already been finalized or on one that had previously returned SQLITE_ERROR or SQLITE_DONE. Or it could be the case that the same database connection is being used by two or more threads at the same moment in time.

- SQLITE_SCHEMA means that the database schema changes, and the SQL statement has been recompiled and run again, but the scheme changed in a way that makes the statement no longer valid, as a fatal error.

- another specific error code is returned on fatal error

**stmt_readonly**: function(S: TSQLite3Statement): integer; cdecl;

*Returns true (non-zero) if and only if the prepared statement X makes no direct changes to the content of the database file*

- Transaction control statements such as BEGIN, COMMIT, ROLLBACK, SAVEPOINT, and RELEASE cause sqlite3.stmt_readonly() to return true, since the statements themselves do not actually modify the database but rather they control the timing of when other statements modify the database. The ATTACH and DETACH statements also cause sqlite3.stmt_readonly() to return true since, while those statements change the configuration of a database connection, they do not make changes to the content of the database files on disk.

**total_changes**: function(DB: TSQLite3DB): Integer; cdecl;

*Total Number Of Rows Modified*

- This function returns the number of row changes caused by INSERT, UPDATE or DELETE statements since the database connection was opened. The count returned by sqlite3.total_changes() includes all changes from all trigger contexts and changes made by foreign key actions. However, the count does not include changes used to implement REPLACE constraints, do rollbacks or ABORT processing, or DROP TABLE processing. The count does not include rows of views that fire an INSTEAD OF trigger, though if the INSTEAD OF trigger makes changes of its own, those changes are counted. The sqlite3.total_changes() function counts the changes as soon as the statement that makes them is completed (when the statement handle is passed to sqlite3.reset() or sqlite3.finalize()).

- If a separate thread makes changes on the same database connection while sqlite3.total_changes() is running then the value returned is unpredictable and not meaningful.
trace_v2: function(DB: TSQLite3DB; Mask: TSQLTraceMask; Callback: TSQLTraceCallback; UserData: Pointer): Pointer; cdecl;

Register callback function that can be used for tracing the execution of SQL statements
- registers a trace callback function Callback against database connection DB, using property mask TSQLTraceMask and context pointer UserData
- if the Callback parameter is nil or if the TSQLTraceMask mask is zero, then tracing is disabled
- parameters of the Callback functions depend of the TSQLTraceMask involved

update_hook: function(DB: TSQLite3DB; xCallback: TSQLUpdateCallback; pArg: pointer): pointer; cdecl;

Register Data Change Notification Callbacks
- The sqlite3.update_hook() interface registers a callback function with the database connection identified by the first argument to be invoked whenever a row is updated, inserted or deleted.
- Any callback set by a previous call to this function for the same database connection is overridden.
- sqlite3.update_hook(D,C,P) function returns the P argument from the previous call on the same database connection D, or nil for the first call on database connection D.
- The update hook is not invoked when internal system tables are modified (i.e. sqlite_master and sqlite_sequence).
- In the current implementation, the update hook is not invoked when duplication rows are deleted because of an ON CONFLICT REPLACE clause. Nor is the update hook invoked when rows are deleted using the truncate optimization. The exceptions defined in this paragraph might change in a future release of SQLite.
- Note that you should also trace COMMIT and ROLLBACK commands (calling sqlite3.commit_hook() and sqlite3.rollback_hook() functions) if you want to ensure that the notified update was not canceled by a later Rollback.

user_data: function(Context: TSQLite3FunctionContext): pointer; cdecl;

Returns a copy of the pointer that was the pUserData parameter (the 5th parameter) of the sqlite3.create_function() routine that originally registered the application defined function
- This routine must be called from the same thread in which the application-defined function is running

value_blob: function(Value: TSQLite3Value): pointer; cdecl;

Converts a sqlite3.value object, specified by its handle, into a blob memory, and returns a copy of that value

value_bytes: function(Value: TSQLite3Value): integer; cdecl;

Number of bytes for a sqlite3.value object, specified by its handle
- used after a call to sqlite3.value_text() or sqlite3.value_blob() to determine buffer size (in bytes)

value_double: function(Value: TSQLite3Value): double; cdecl;

Converts a sqlite3.value object, specified by its handle, into a floating point value and returns a copy of that value

value_int64: function(Value: TSQLite3Value): Int64; cdecl;

Converts a sqlite3.value object, specified by its handle, into an integer value and returns a copy of that value
### Synopsis ORMot Framework

**Software Architecture Design 1.18**

**Date:** September 16, 2020

---

**value_numeric_type:**

```delphi
function (Value: TSQLite3Value): integer; cdecl;
```

*Attempts to apply numeric affinity to the value*

- This means that an attempt is made to convert the value to an integer or floating point. If such a conversion is possible without loss of information (in other words, if the value is a string that looks like a number) then the conversion is performed. Otherwise no conversion occurs. The datatype after conversion is returned.
- returned value is one of SQLITE_INTEGER, SQLITE_FLOAT, SQLITE_TEXT, SQLITE_BLOB or SQLITE_NULL

**value_text:**

```delphi
function (Value: TSQLite3Value): PUTF8Char; cdecl;
```

*Converts a sqlite3.value object, specified by its handle, into an UTF-8 encoded string, and returns a copy of that value*

**value_type:**

```delphi
function (Value: TSQLite3Value): integer; cdecl;
```

*Datatype code for a sqlite3.value object, specified by its handle*

- returned value is one of SQLITE_INTEGER, SQLITE_FLOAT, SQLITE_TEXT, SQLITE_BLOB or SQLITE_NULL
- must be called before any sqlite3.value_*() statement, which may result in an implicit type conversion: in this case, value is undefined

**constructor Create; virtual;**

*Initialize the internal version numbers*

**procedure ForceToUseSharedMemoryManager; virtual;**

*Will change the SQLite3 configuration to use Delphi/FPC memory manager*

- this will reduce memory fragmentation, and enhance speed, especially under multi-process activity
- this method should be called before sqlite3.initialize()

**property Version:**

```delphi
RawUTF8 read GetVersion;
```

*Will return the class name and SQLite3 version number*

- if self (e.g. global sqlite3) is nil, will return ""

**property VersionNumber:**

```delphi
cardinal read fVersionNumber;
```

*Returns the current version number as a plain integer*

- equals e.g. 3008003001 for '3.8.3.1'

**property VersionText:**

```delphi
RawUTF8 read fVersionText;
```

*Returns the current version number as a text*

- equals e.g. '3.8.3.1'
- use the Version property for the full information about this instance

---

**TSQLite3LibraryDynamic = class(TSQLite3Library)**

*Allow access to an external SQLite3 library engine*

- you can e.g. replace the main sqlite3 engine with any external library:

```delphi
FreeAndNil(sqlite3); // release any previous instance (e.g. static)
sqlite3 := TSQLite3LibraryDynamic.Create;
```

*Used for DI-2.2.1 (page 2548).*
constructor Create(const LibraryName: TFileName=SQLITE_LIBRARY_DEFAULT_NAME);
reintroduce;
  Initialize the specified external library
  - raise an ESQLite3Exception on error

destructor Destroy; override;
  Unload the external library

ESQLite3Exception = class(ESynException)
Custom SQLite3 dedicated Exception type
  DB: TSQLite3DB;
  The DB which raised this exception

constructor Create(aDB: TSQLite3DB; aErrorCode: integer; const aSQL: RawUTF8);
reintroduce; overload;
  Create the exception, getting the message from DB

property ErrorCode: integer read fErrorCode;
  The corresponding error code, e.g. 21 (for SQLITE_MISUSE)

property SQLite3ErrorCode: TSQLite3ErrorCode read fSQLite3ErrorCode;
  The corresponding error code, e.g. secMISUSE

TSQLRequest = object(TObject)
Wrapper to a SQLite3 request
  - defined as a record, so that it may be allocated on the stack
  - do not forget to call the Close method to release the request resources

Used for DI-2.2.1 (page 2548).

function Execute(aDB: TSQLite3DB; const aSQL: RawUTF8; var Values: TRawUTF8DynArray): integer; overload;
  Execute a SQL statement which return TEXT from the aSQL UTF-8 encoded string
  - Execute the first statement in aSQL
  - this statement must get (at least) one field/column result of TEXT
  - return result as a dynamic array of RawUTF8 in ID
  - return count of row in integer function result (may be < length(ID))
  - raise an ESQLite3Exception on any error

Used for DI-2.2.1 (page 2548).

function Execute(aDB: TSQLite3DB; const aSQL: RawUTF8; var ID: TInt64DynArray): integer; overload;
  Execute a SQL statement which return integers from the aSQL UTF-8 encoded string
  - Execute the first statement in aSQL
  - this statement must get (at least) one field/column result of INTEGER
  - return result as a dynamic array of Int64 in ID
  - return count of row in integer function result (may be < length(ID))
  - raise an ESQLite3Exception on any error

Used for DI-2.2.1 (page 2548).
function Execute(aDB: TSQLite3DB; const aSQL: RawUTF8; JSON: TStream; Expand: boolean=false): PTrInt; overload;

Execute one SQL statement which return the results in JSON format
- JSON format is more compact than XML and well supported
- Execute the first statement in aSQL
- if SQL is '', the statement should have been prepared, reset and bound if necessary
- raise an ESQLite3Exception on any error
- JSON data is added to TStream, with UTF-8 encoding
- if Expand is true, JSON data is an array of objects, for direct use with any Ajax or .NET client:
  [ { "col1":val11,"col2": "val12"},{"col1":val21,... } ]
- if Expand is false, JSON data is serialized (used in TSQLTableJSON)
  { "FieldCount":1,"Values":["col1","col2",val11,"val12",val21,..] }

- BLOB field value is saved as Base64, in the "\uFFFF0base64encodedbinary" format and contains true BLOB data (no conversion into TEXT, as with TSQLiteTableDB) - so will work for sftBlob, sftBlobDynArray and sftBlobRecord
- returns the number of data rows added to JSON (excluding the headers)

Used for DI-2.2.1 (page 2548).

function ExecuteJSON(aDB: TSQLite3DB; const aSQL: RawUTF8; Expand: boolean=false; aResultCount: PPtrInt=nil): RawUTF8;

Execute one SQL statement which return the results in JSON format
- use internaly Execute() above with a TRawByteStringStream, and return a string
- BLOB field value is saved as Base64, e.g. "\uFFFF0base64encodedbinary"
- returns the number of data rows added to JSON (excluding the headers) in the integer variable mapped by aResultCount (if any)
- if any error occurs, the ESQLite3Exception is handled and '' is returned

function ExecuteNoException(aDB: TSQLite3DB; const aSQL: RawUTF8): boolean;

Execute one SQL statement in the aSQL UTF-8 encoded string
- Execute the first statement in aSQL: call Prepare() then Step once
- Close is always called internaly
- returns TRUE on success, and raise no ESQLite3Exception on error, but returns FALSE

function FieldA(Col: integer): WinAnsiString;

Return a field as Win-Ansi (i.e. code page 1252) encoded text value, first Col is 0

function FieldBlob(Col: integer): RawByteString;

Return a field as a blob value (RawByteString/TSQLRawBlob is an AnsiString), first Col is 0

function FieldBlobToStream(Col: integer): TStream;

Return a field as a TStream blob value, first Col is 0
- caller shall release the returned TStream instance

function FieldDeclaredType(Col: Integer): RawUTF8;

Return the type of this column, as declared at creation
- textual type used for CREATE TABLE of the corresponding column, as returned by sqlite3.column_decltype()
function FieldDeclaredTypeS(Col: Integer): string;
    Return the generic VCL string type of this column, as declared at creation
    - textual type used for CREATE TABLE of corresponding column, as returned by
      sqlite3.column_decltype()
    - note that prior to Delphi 2009, you may loose content during conversion

function FieldDouble(Col: integer): double;
    Return a field floating point value, first Col is 0

function FieldIndex(const aColumnName: RawUTF8): integer;
    The field index matching this name
    - return -1 if not found

function FieldInt(Col: integer): Int64;
    Return a field integer value, first Col is 0

function FieldName(Col: integer): RawUTF8;
    3. Field attributes after a sucessfull Step() (returned SQLITE_ROW) the field name of the current
       ROW

function FieldNull(Col: Integer): Boolean;
    Return TRUE if the column value is NULL, first Col is 0

function Fields(Col: integer): string;
    Return a text value value as generic VCL string, first Col is 0
    - note that prior to Delphi 2009, you may loose content during conversion

function FieldType(Col: Integer): integer;
    Return the field type of this column
    - retrieve the "SQLite3" column type as returned by sqlite3.column_type - i.e. SQLITE_NULL,
      SQLITE_INTEGER, SQLITE_FLOAT, SQLITE_TEXT, or SQLITE_BLOB

function FieldUTF8(Col: integer): RawUTF8;
    Return a field UTF-8 encoded text value, first Col is 0

function FieldValue(Col: integer): TSQLite3Value;
    Return the field as a sqlite3.value object handle, first Col is 0

function FieldW(Col: integer): RawUnicode;
    Return a field RawUnicode encoded text value, first Col is 0

function Prepare(DB: TSQLite3DB; const SQL: RawUTF8; NoExcept: boolean=false): integer;
    1. general request process Prepare a UTF-8 encoded SQL statement
       - compile the SQL into byte-code
       - parameters ? ?NNN :VV @VV $VV can be bound with Bind*() functions below
       - raise an ESQLite3Exception on any error, unless NoExcept is TRUE

function PrepareAnsi(DB: TSQLite3DB; const SQL: WinAnsiString): integer;
    Prepare a WinAnsi SQL statement
    - behave the same as Prepare()
function PrepareNext: integer;
   Prepare the next SQL command initialized in previous Prepare()
   - raise an ESQLite3Exception on any error

function Reset: integer;
   Reset A Prepared Statement Object
   - reset a prepared statement object back to its initial state, ready to be re-executed.
   - any SQL statement variables that had values bound to them using the Bind*() function below retain their values. Use BindReset() to reset the bindings
   - return SQLITE_OK on success, or the previous Step error code

function Step: integer;
   Evaluate An SQL Statement, returning the sqlite3.step() result status:
   - return SQLITE_ROW on success, with data ready to be retrieved via the Field*() methods
   - return SQLITE_DONE if the SQL commands were executed
   - raise an ESQLite3Exception on any error

procedure Bind(Param: Integer; Data: TCustomMemoryStream); overload;
   Bind a Blob TCustomMemoryStream buffer to a parameter
   - the leftmost SQL parameter has an index of 1, but ?NNN may override it
   - raise an ESQLite3Exception on any error

procedure Bind(Param: Integer; Value: double); overload;
   Bind a double value to a parameter
   - the leftmost SQL parameter has an index of 1, but ?NNN may override it
   - raise an ESQLite3Exception on any error

procedure Bind(Param: Integer; const Value: RawUTF8); overload;
   Bind a UTF-8 encoded string to a parameter
   - the leftmost SQL parameter has an index of 1, but ?NNN may override it
   - raise an ESQLite3Exception on any error
   - this function will use copy-on-write assignment of Value, with no memory allocation, then let sqlite3InternalFreeRawByteString release the variable

procedure Bind(Param: Integer; Data: pointer; Size: integer); overload;
   Bind a Blob buffer to a parameter
   - the leftmost SQL parameter has an index of 1, but ?NNN may override it
   - raise an ESQLite3Exception on any error

procedure Bind(Param: Integer; Value: Int64); overload;
   Bind an integer value to a parameter
   - the leftmost SQL parameter has an index of 1, but ?NNN may override it
   - raise an ESQLite3Exception on any error

procedure BindBlob(Param: Integer; const Data: RawByteString);
   Bind a Blob buffer to a parameter
   - the leftmost SQL parameter has an index of 1, but ?NNN may override it
   - raise an ESQLite3Exception on any error
   - this function will use copy-on-write assignment of Data, with no memory allocation, then let sqlite3InternalFreeRawByteString release the variable
procedure BindNull(Param: Integer);

Bind a NULL value to a parameter
- the leftmost SQL parameter has an index of 1, but ?NNN may override it
- raise an ESQLite3Exception on any error

procedure BindReset;

2. Bind parameters to a SQL query (for the last prepared statement) Reset All Bindings On A Prepared Statement
- Contrary to the intuition of many, Reset() does not reset the bindings on a prepared statement. Use this routine to reset all host parameter

procedure BindS(Param: Integer; const Value: string);

Bind a generic VCL string to a parameter
- with versions prior to Delphi 2009, you may loose some content here: Bind(Param: integer; Value: RawUTF8) is the prefered method
- the leftmost SQL parameter has an index of 1, but ?NNN may override it
- raise an ESQLite3Exception on any error

procedure BindZero(Param: Integer; Size: integer);

Bind a ZeroBlob buffer to a parameter
- uses a fixed amount of memory (just an integer to hold its size) while it is being processed.
  Zeroblobs are intended to serve as placeholders for BLOBs whose content is later written using incremental BLOB I/O routines (as with TSQLBlobStream created from TSQLDataBase.Blob() e.g.).
- a negative value for the Size parameter results in a zero-length BLOB
- the leftmost SQL parameter has an index of 1, but ?NNN may override it
- raise an ESQLite3Exception on any error

procedure Close;

Close the Request handle
- call it even if an ESQLite3Exception has been raised

procedure Execute(aDB: TSQLite3DB; const aSQL: RawUTF8); overload;

Execute one SQL statement in the aSQL UTF-8 encoded string
- Execute the first statement in aSQL: call Prepare() then Step once
- Close is always called internaly
- raise an ESQLite3Exception on any error

Used for DI-2.2.1 (page 2548).

procedure Execute; overload;

Execute one SQL statement already prepared by a call to Prepare()
- the statement is closed
- raise an ESQLite3Exception on any error

Used for DI-2.2.1 (page 2548).
procedure Execute(aDB: TSQLite3DB; const aSQL: RawUTF8; out ID: Int64); overload;

Execute a SQL statement which return one integer from the aSQL UTF-8 encoded string
- Execute the first statement in aSQL
- this statement must get (at least) one field/column result of INTEGER
- return result as an unique Int64 in ID
- raise an ESQLite3Exception on any error }

Used for DI-2.2.1 (page 2548).

procedure Execute(aDB: TSQLite3DB; const aSQL: RawUTF8; out Value: RawUTF8); overload;

Execute a SQL statement which return one TEXT value from the aSQL UTF-8 encoded string
- Execute the first statement in aSQL
- this statement must get (at least) one field/column result of TEXT
- raise an ESQLite3Exception on any error

Used for DI-2.2.1 (page 2548).

procedure ExecuteAll; overload;

Execute all SQL statements already prepared by a call to Prepare()
- the statement is closed
- raise an ESQLite3Exception on any error

procedure ExecuteAll(aDB: TSQLite3DB; const aSQL: RawUTF8); overload;

Execute all SQL statements in the aSQL UTF-8 encoded string
- interanly call Prepare() then Step then PrepareNext until end of aSQL
- Close is always called interanly
- raise an ESQLite3Exception on any error

procedure ExecuteDebug(aDB: TSQLite3DB; const aSQL: RawUTF8; var OutFile: Text);

Execute all SQL statements in the aSQL UTF-8 encoded string, results will be written as ANSI text in OutFile

procedure FieldsToJSON(WR: TJSONWriter; DoNotFetchBlobs: boolean=false);

Append all columns values of the current Row to a JSON stream
- will use WR.Expand to guess the expected output format
- BLOB field value is saved as Base64, in the "\uFFF0base64encodedbinary" format and contains true BLOB data

property FieldCount: integer read fFieldCount;

The column/field count of the current ROW
- fields numerotation starts with 0

property IsReadOnly: Boolean read GetReadOnly;

Returns true if the current prepared statement makes no direct changes to the content of the database file
- Transaction control statements such as BEGIN, COMMIT, ROLLBACK, SAVEPOINT, and RELEASE cause this property to return true, since the statements themselves do not actually modify the database but rather they control the timing of when other statements modify the database. The ATTACH and DETACH statements also cause this property to return true since, while those statements change the configuration of a database connection, they do not make changes to the content of the database files on disk.
property ParamCount: integer read GetParamCount;
The bound parameters count

property Request: TSQLite3Statement read fRequest;
Read-only access to the Request (SQLite3 statement) handle

property RequestDB: TSQLite3DB read fDB;
Read-only access to the SQLite3 database handle

TSQLStatementCache = record
  Used to retrieve a prepared statement
  Statement: TSQLRequest;
    Associated prepared statement, ready to be executed after binding
  StatementSQL: RawUTF8;
    Associated SQL statement
  Timer: TSynMonitor;
    Used to monitor execution time
end;

TSQLStatementCached = object(TObject)
  Handle a cache of prepared statements
  - is defined either as an object either as a record, due to a bug in Delphi 2009/2010 compiler (at least): this structure is not initialized if defined as an object on the stack, but will be as a record :(
  Cache: TSQLStatementCacheDynArray;
    Prepared statements with parameters for faster SQLite3 execution
    - works for SQL code with ? internal parameters
  Caches: TDynArrayHashed;
    Hashing wrapper associated to the Cache[] array
  Count: integer;
    Current number of items in the Cache[] array
  DB: TSQLite3DB;
    The associated SQLite3 database instance
    - any direct access to this cache list should be protected via DB.Lock
  function Prepare(const GenericSQL: RawUTF8; WasPrepared: PBoolean=nil;
    ExecutionTimer: PPPrecisionTimer=nil; ExecutionMonitor: PSynMonitor=nil): PSQLRequest;
    Add or retrieve a generic SQL (with ? parameters) statement from cache
  procedure Init(aDB: TSQLite3DB);
    Initialize the cache
  procedure ReleaseAllDBStatements;
    Used internally to release all prepared statements from Cache[]
procedure SortCacheByTotalTime(var aIndex: TIntegerDynArray);

Could be used e.g. for statistics
- will use internally the function StatementCacheTotalTimeCompare()

TSQLDataBaseSQLFunction = class(TObject)
Those classes can be used to define custom SQL functions inside a TSQLDataBase

constructor Create(aFunction: TSQLFunctionFunc; aFunctionParametersCount: Integer; const aFunctionName: RawUTF8=''); reintroduce;

Initialize the corresponding SQL function
- expects at least the low-level TSQLFunctionFunc implementation (in sqlite3.create_function() format) and the number of expected parameters
- if the function name is not specified, it will be retrieved from the type information (e.g. TReferenceDynArray will declare 'ReferenceDynArray')

property FunctionName: RawUTF8 read fSQLName;
The SQL function name, as called from the SQL statement
- the same function name may be registered several times with a diverse number of parameters (e.g. to implement optional parameters)

property FunctionParametersCount: integer read fFunctionParametersCount;
The number of parameters expected by the SQL function

property InternalFunction: TSQLFunctionFunc read fInternalFunction;
The internal function prototype
- ready to be assigned to sqlite3.create_function() xFunc parameter

TSQLDataBaseSQLFunctionDynArray = class(TSQLDataBaseSQLFunction)
To be used to define custom SQL functions for dynamic arrays BLOB search

constructor Create(aTypeInfo: pointer; aCompare: TDynArraySortCompare; const aFunctionName: RawUTF8=''); reintroduce;

Initialize the corresponding SQL function
- if the function name is not specified, it will be retrieved from the type information (e.g. TReferenceDynArray will declare 'ReferenceDynArray')
- the SQL function will expect two parameters: the first is the BLOB field content, and the 2nd is the array element to search (set with TDynArray.ElemSave() or with BinToBase64WithMagic(aDynArray.ElemSave()) if called via a Client and a JSON prepared parameter)
- you should better use the already existing faster SQL functions Byte/Word/Integer/Cardinal/Int64/CurrencyDynArrayContains() if possible (this implementation will allocate each dynamic array into memory before comparison, and will be therefore slower than those optimized versions)
**TSQLDatabase** = class(TSynPersistentLock)

*Simple wrapper for direct SQLite3 database manipulation*
- embed the SQLite3 database calls into a common object
- thread-safe call of all SQLite3 queries (SQLITE_THREADSAFE 0 in sqlite.c)
- can cache last results for SELECT statements, if property UseCache is true: this can speed up most read queries, for web server or client UI e.g.

*Used for DI-2.2.1 (page 2548).*

**constructor** Create(const aFileName: TFileName; const aPassword: RawUTF8=''; aOpenV2Flags: integer=0; aDefaultCacheSize: integer=10000; aDefaultPageSize: integer=4096); reintroduce;

*Open a SQLite3 database file*
- open an existing database file or create a new one if no file exists
- if specified, the password will be used to cypher this file on disk (the main SQLite3 database file is encrypted, not the wal file during run); the password may be a JSON-serialized TSynSignerParams object, or will use AES-OFB-128 after SHAKE_128 with rounds=1000 and a fixed salt on plain password text; note that our custom encryption is not compatible with the official SQLite Encryption Extension module
- you can specify some optional flags for sqlite3.open_v2() as SQLITE_OPEN_READONLY or SQLITE_OPEN_READWRITE instead of supplied default value (which corresponds to the sqlite3.open() behavior)
- by default, 10000 pages are used to cache data in memory (using around 40 MB of RAM), but you may specify another value for performance tuning
- SYSTEMNOCASE collation is added (our custom fast UTF-8 case insensitive UTF8ILComp() function, which is used also in the SQLite3UI unit for coherency and efficiency
- ISO8601 collation is added (TDatetime stored as ISO-8601 encoded TEXT)
- WIN32CASE and WIN32NOCASE collations are added (use slow but accurate Win32 CompareW)
- some additional SQL functions are registered: MOD, SOUNDEX/SOUNDEXFR/SOUNDEXES, RANK, CONCAT, TIMELOG, TIMELOGUNIX, JSONGET/JSONHAS/JSONSET and TDynArray-Blob Byte/Word/Integer/Cardinal/Int64/Currency/RawUTF8DynArrayContains
- initialize an internal mutex to ensure that all access to the database is atomic
- raise an ESQLite3Exception on any error

**destructor** Destroy; override;

*Close a database and free its memory and context*
- if TransactionBegin was called but not committed, a RollBack is performed

**function** Backup(const BackupFileName: TFileName): boolean;

*Backup of the opened Database into an external file name*
- warning: this method won't use the SQLite Online Backup API
- database is closed, VACCUUMed, copied, then reopened: it's very fast for small databases, but is blocking and should be an issue
- if you use some virtual tables, they won't be restored after backup: this method would probably fail e.g. in the context of mORMot.pas
function BackupBackground(const BackupFileName: TFileName; StepPageNumber, StepSleepMS: Integer; OnProgress: TSQLDatabaseBackupEvent; SynLzCompress: boolean=false; const aPassword: RawUTF8=''): boolean;

Backup of the opened Database into an external file name
- this method will use the SQLite Online Backup API and a dedicated background thread for the process
- this will be asynchronous, and would block the main database process only when copying the StepPageNumber number of pages for each step, waiting StepSleepMS milliseconds before performing the next step: as a result, the copy operation can be done incrementally, by blocks of StepPageNumber pages, in which case the source database does not need to be locked for the duration of the copy, only for the brief periods of time when it is actually being read from: this allows other database users to continue uninterrupted (at least during StepSleepMS milliseconds) while a backup is running
- if StepPageNumber is -1, the whole DB will be copied in a single step, therefore in blocking mode, e.g. with BackupBackgroundWaitUntilFinished
- if SynLzCompress is TRUE, the backup file would be compressed using FileSynLZ() function - you may use BackupUnSynLZ() class method to uncompress the .dbsynlz file into a proper SQLite3 file
- the supplied OnProgress event handler will be called at each step, in the context of the background thread
- the background thread will be released when the process is finished
- if only one connection to the database does exist (e.g. if you use only one TSQLDataBase instance on the same database file), any modification to the source database during the background process will be included in the backup - so this method will work perfectly e.g. for mORMot.pas
- if specified, a password will be used to cypher BackupFileName on disk (it will work only with SynSQLite3Static) - you can uncypher the resulting encrypted database file later via ChangeSQLEncryptTablePassWord()
- returns TRUE if backup started as expected, or FALSE in case of error (e.g. if there is already another backup started, if the source or destination databases are locked or invalid, or if the sqlite3.dll is too old and does not support the Online Backup API)
- you can also use this method to save an SQLite3 ':memory:' database, perhaps in conjunction with the BackupBackgroundWaitUntilFinished method

function BackupBackgroundToDB(BackupDB: TSQLDatabase; StepPageNumber, StepSleepMS: Integer; OnProgress: TSQLDatabaseBackupEvent): boolean;

Background backup to another opened database instance
- in respect to BackupBackground method, it will use an existing database the actual process
- by design, SynLZCompress or aPassword parameters are unavailable

class function BackupSynLZ(const SourceDB, DestSynLZ: TFileName; EraseSourceDB: boolean): boolean;

Compress a SQLite3 file into a proprietary but efficient .dbsynlz layout
- same format than BackupUnSynLZ() class method or if SynLZCompress parameter is TRUE for BackupBackground() method
- the SourceDB file should not be active (e.g. be a backup file), i.e. not currently opened by the SQLite3 engine, otherwise behavior is unknown
- returns TRUE on success, FALSE on failure
class function BackupUnSynLZ(const SourceSynLZ, DestDB: TFileName): boolean;

Uncompress a .dbsynlz backup file as previously compressed with BackupSynLZ() or if SynLZCompress parameter is TRUE for BackupBackground() method
- any DestDB file name would be overwritten
- returns TRUE on success, FALSE on failure

function Blob(const DBName, TableName, ColumnName: RawUTF8; RowID: Int64; ReadWrite: boolean=false): TSQLBlobStream;

Open a BLOB incrementally for read/write access
- find a BLOB located in row RowID, column ColumnName, table TableName in database
- use after a TSQLRequest.BindZero() to reserve Blob memory
- if RowID=0, then the last inserted RowID is used (beware that this value won't be thread-safe, if another thread run another INSERT)
- will raise an ESQLite3Exception on any error

function DBClose: integer;

Close the opened database
- TSQLDatabase.Destroy already closes the database: this method is to be used only on particular cases, e.g. to close temporary a DB file and allow making a backup on its content
- returns the SQLITE_* status code, as retrieved from sqlite3.close(fDB) so that it should be SQLITE_OK on success

function DBOpen: integer; virtual;

(re)open the database from file fFileName
- TSQLDatabase.Create already opens the database: this method is to be used only on particular cases, e.g. to close temporary a DB file and allow making a backup on its content
- returns the SQLITE_* status code, as retrieved from sqlite3.open() so that it should be SQLITE_OK on success

function EnableCustomTokenizer: integer;

For SQLite >= 3.11 - enable registration of a custom tokenizer
- see details at http://sqlite.org/fts3.html#f3tknzr

function Execute(const aSQL: RawUTF8; var Values: TRawUTF8DynArray): integer; overload;

Execute one SQL statement returning TEXT from the aSQL UTF-8 encoded string
- Execute the first statement in aSQL
- this statement must get (at least) one field/column result of TEXT
- return result as a dynamic array of RawUTF8 in ID
- return count of row in integer function result (may be < length(ID))
- raise an ESQLite3Exception on any error

function Execute(const aSQL: RawUTF8; var ID: TInt64DynArray): integer; overload;

Execute one SQL statement which return integers from the aSQL UTF-8 encoded string
- Execute the first statement in aSQL
- this statement must get a one field/column result of INTEGER
- return result as a dynamic array of RawUTF8, as TEXT result
- return count of row in integer function result (may be < length(ID))
- raise an ESQLite3Exception on any error
function ExecuteJSON(const aSQL: RawUTF8; Expand: boolean=false; aResultCount: PPtrInt=nil): RawUTF8;

    Execute one SQL statement returning its results in JSON format
    - the BLOB data is encoded as "\uFFF0base64encodedbinary"

function ExecuteNoException(const aSQL: RawUTF8): boolean;

    Execute one SQL statements in an UTF-8 encoded string
    - can be prepared with TransactionBegin()
    - raise no Exception on error, but returns FALSE in such case

function ExecuteNoExceptionInt64(const aSQL: RawUTF8): Int64;

    Seamless execution of a SQL statement which returns one integer
    - Execute the first statement in aSQL
    - this statement must get a one field/column result of INTEGER
    - returns 0 on any error

function ExecuteNoExceptionUTF8(const aSQL: RawUTF8): RawUTF8;

    Seamless execution of a SQL statement which returns one UTF-8 encoded string
    - Execute the first statement in aSQL
    - this statement must get a one field/column result of TEXT
    - returns " " on any error

function HasTable(const Name: RawUTF8): boolean;

    Check if the given table do exist

class function IsBackupSyncLZFile(const SynLZFile: TFileName): boolean;

    Returns TRUE if the supplied name is a SQLite3 .dbsynlz compressed file
    - i.e. on the format generated by the BackupUnSyncLZ() class method or if SynLZCompress parameter is TRUE for BackupBackground() method

function LastChangeCount: integer;

    Count the number of rows modified by the last SQL statement
    - this method returns the number of database rows that were changed or inserted or deleted by the most recently completed SQL statement on the database connection specified by the first parameter. Only changes that are directly specified by the INSERT, UPDATE, or DELETE statement are counted.
    - wrapper around the sqlite3.changes() low-level function

function LastInsertRowID: Int64;

    Return the last insert rowid

function LockJSON(const aSQL: RawUTF8; aResultCount: PPtrInt): RawUTF8;

    Enter the internal mutex: called before any DB access
    - provide the SQL statement about to be executed: handle proper caching
    - if this SQL statement has an already cached JSON response, return it and don't enter the internal mutex: no UnLockJSON() call is necessary
    - if this SQL statement is not a SELECT, cache is flushed and the next call to UnLockJSON() won't add any value to the cache since this statement is not a SELECT and doesn't have to be cached!
    - if aResultCount does map to an integer variable, it will be filled with the returned row count of data (excluding field names) in the result
function TotalChangeCount: integer;
  Return the number of row changes caused by INSERT, UPDATE or DELETE statements since the
database connection was opened
- wrapper around the sqlite3.total_changes() low-level function

procedure BackupBackgroundWaitUntilFinished(TimeOutSeconds: Integer=-1);
  Wait until any previous BackupBackground() is finished
- warning: this method won’t call the Windows message loop, so should not be called from main
thread, unless the UI may become unresponsive: you should better rely on OnProgress() callback
for any GUI application
- by default, it will wait for ever so that process is finished, but you can set a time out (in
seconds) after which the process will be aborted
- could be used with BackupBackground() and StepPageNumber=-1 to perform a whole copy of a
database in one shot:
  if aDB.BackupBackground('backup.db3',-1,0,nil) then
  aDB.BackupBackgroundWaitUntilFinished;

procedure CacheFlush;
  Flush the internal SQL-based JSON cache content
- to be called when the regular Lock/LockJSON methods are not called, e.g. with external tables
as defined in SQLite3DB unit
- will also increment the global InternalState property value (if set)

procedure Commit;
  End a transaction: write all Execute() statements to the disk

procedure Execute(const aSQL: RawUTF8; out ID: Int64; NoLog: boolean=false);
  overload;
  Execute one SQL statement which returns one integer from the aSQL UTF-8 encoded string
  - Execute the first statement in aSQL
  - this statement must get a one field/column result of INTEGER
  - raise an ESQLite3Exception on any error

procedure Execute(const aSQL: RawUTF8); overload;
  Execute one SQL statements in aSQL UTF-8 encoded string
  - can be prepared with TransactionBegin()
  - raise an ESQLite3Exception on any error

procedure Execute(const aSQL: RawUTF8; out ID: RawUTF8; NoLog: boolean=false);
  overload;
  Execute one SQL statement which returns one UTF-8 encoded string value
  - Execute the first statement in aSQL
  - this statement must get a one field/column result of TEXT
  - raise an ESQLite3Exception on any error

procedure ExecuteAll(const aSQL: RawUTF8);
  Execute all SQL statements in aSQL UTF-8 encoded string
  - can be prepared with TransactionBegin()
  - raise an ESQLite3Exception on any error

procedure GetFieldNames(var Names: TRawUTF8DynArray; const TableName: RawUTF8);
  Get all field names for a specified Table
procedure GetTableNames(var Names: TRawUTF8DynArray);
  Get all table names contained in this database file

procedure Lock; overload;
  Enter the internal mutex without any cache flush
  - same as Lock('');

procedure Lock(const aSQL: RawUTF8); overload;
  Class function = bug in D2005 enter the internal mutex: called before any DB access
  - provide the SQL statement about to be executed: handle proper caching
  - if the SQL statement is void, assume a SELECT statement (no cache flush)

procedure LockAndFlushCache;
  Flush the internal statement cache, and enter the internal mutex
  - same as Lock('ALTER');

procedure RegisterSQLFunction(aDynArrayTypeInfo: pointer; aCompare: TDynArraySortCompare; const aFunctionName: RawUTF8=''); overload;
  Add a SQL custom function for a dynamic array to the database
  - the resulting SQL function will expect two parameters: the first is the BLOB field content, and
  the 2nd is the array element to search (as set with TDynArray.ElemSave() or with
  BinToBase64WithMagic(aDynArray.ElemSave()) if called via a Client and a JSON prepared
  parameter)
  - if the function name is not specified, it will be retrieved from the type information (e.g.
  TReferenceDynArray will declare 'ReferenceDynArray')
  - you should better use the already existing faster SQL functions
    Byte/Word/Integer/Cardinal/Int64/CurrencyDynArrayContains() if possible (this implementation
    will allocate each dynamic array into memory before comparison, and will be therefore slower
    than those optimized versions - but it will be always faster than Client-Server query, in all cases)

procedure RegisterSQLFunction(aFunction: TSQLFunctionFunc; aFunctionParametersCount: Integer; const aFunctionName: RawUTF8); overload;
  Add a SQL custom function to the SQLite3 database engine
  - will do nothing if the same function name and parameters count have already been registered
    (you can register then same function name with several numbers of parameters)
  - typical use may be:
    Demo.RegisterSQLFunction(InternalSQLFunctionCharIndex,2,'CharIndex');

procedure RegisterSQLFunction(aFunction: TSQLDataBaseSQLFunction); overload;
  Add a SQL custom function to the SQLite3 database engine
  - the supplied aFunction instance will be used globally and freed by TSQLDataBase.Destroy
  destructor
  - will do nothing if the same function name and parameters count have already been registered
    (you can register then same function name with several numbers of parameters)
  - you may use the overloaded function, which is a wrapper around:
    Demo.RegisterSQLFunction(
      TSQLDataBaseSQLFunction.Create(InternalSQLFunctionCharIndex,2,'CharIndex'));

procedure RollBack;
  Abort a transaction: restore the previous state of the database
procedure TransactionBegin(aBehavior: TSQLDataBaseTransactionBehaviour = tbDeferred);

Begin a transaction
- Execute SQL statements with Execute() procedure below
- must be ended with Commit on success
- must be aborted with Rollback after an ESQLite3Exception raised
- The default transaction behavior is tbDeferred

procedure UnLock;

Leave the internal mutex: called after any DB access

procedure UnLockJSON(const aJSONResult: RawUTF8; aResultCount: PtrInt);

Leave the internal mutex: called after any DB access
- caller must provide the JSON result for the SQL statement previously set by LockJSON()
- do proper caching of the JSON response for this SQL statement

property BackupBackgroundInProcess: boolean read GetBackupBackgroundInProcess;

Is set to TRUE while a BackupBackground() process is still running
- see also BackupBackgroundWaitUntilFinished() method

property BackupBackgroundLastFileName: TFileName read fBackupBackgroundLastFileName;

The latest BackupBackground() process file name

property BackupBackgroundLastTime: RawUTF8 read fBackupBackgroundLastTime;

How much time did the latest BackupBackground() finished process take

property BusyTimeout: Integer read fBusyTimeout write SetBusyTimeout;

Sets a busy handler that sleeps for a specified amount of time (in milliseconds) when a table is locked, before returning an error

property Cache: TSynCache read fCache;

Access to the internal JSON cache, used by ExecuteJSON() method
- see UseCache property and CacheFlush method

property CacheSize: cardinal read GetCacheSize write SetCacheSize;

Query or change the suggested maximum number of database disk pages that SQLite will hold in memory at once per open database file
- DBOpen method will set this cache size to a big 10000 default, which sounds reasonnable in the context of a server application (will use up to 40 MB of memory cache, with the default PageSize of 4096 bytes)
- when you change the cache size using the cache_size pragma, the change only endures for the current session. The cache size reverts to the default value when the database is closed and reopened
- we do not handle negative values here (i.e. KB of RAM), since it won't work if the linked SQLite3 library is version 3.7.9 and earlier

property DB: TSQLite3DB read fDB;

Read-only access to the SQLite3 database handle

property FileName: TFileName read fFileName;

Read-only access to the SQLite3 database filename opened
property FileNameWithoutPath: TFileName read FileNameWithoutPath;
  Read-only access to the SQLite3 database filename opened without its path

property FileSize: Int64 read GetFileSize;
  Return the total number of bytes in the database file
  - computes PageSize*PageCount

property InternalState: PCardinal read fInternalState write fInternalState;
  This integer pointer (if not nil) is incremented when any SQL statement changes the database contents (i.e. any not SELECT statement)
  - this pointer is thread-safe updated, inside a critical section

property IsMemory: boolean read fIsMemory;
  Equals TRUE if the SQLite3 database was created as ':memory:' (i.e. SQLITE_MEMORY_DATABASE_NAME)

property Limit[Category: TSQLLimitCategory]: integer read GetLimit write SetLimit;
  Retrieve of define a limit on the current database connection
  - see TSQLLimitCategory for a details of all available limits
  - see @http://www.sqlite.org/c3ref/limit.html

property LockingMode: TSQLLockingMode read GetLockingMode write SetLockingMode;
  Query or change the SQLite3 file-based locking mode, i.e. the way it locks the file
  - default lmNormal is ACID and safe
  - lmExclusive gives better performance in case of a number of write transactions, so can be used to release a mORMot server power: but you won't be able to access the database file from outside the process (like a "normal" database engine)

property Log: TSynLogClass read fLog write fLog;
  Access to the log class associated with this SQLite3 database engine
  - can be customized, e.g. by overridden TSQLRestServerDB.SetLogClass()

property LogResultMaximumSize: integer read fLogResultMaximumSize write fLogResultMaximumSize;
  Sets a maximum size (in bytes) to be logged as sqlite rows
  - by default, is set to 512 bytes, which sounds a good compromise since it does not make sense to log all the JSON content retrieved from the database engine, when a huge SELECT is executed

property MemoryMappedMB: cardinal read GetMemoryMappedMB write SetMemoryMappedMB;
  Enables or disables disk content access using memory-mapped I/O
  - 0 to disable it (the default, because of potential disadvantages)
  - set to a number of Mega Bytes value of memory for the mapping
  - expects a SQLite3 engine version >= 3.7.17
  - Memory-Mapped I/O is NOT compatible with password encryption as implemented in our SynSQLite3Static unit

property OpenV2Flags: Integer read fOpenV2Flags;
  Reflects how the database connection was created in the constructor

property PageCount: cardinal read GetPageCount;
  Return the total number of pages in the database file
property PageSize: cardinal read GetPageSize write SetPageSize;

Query or change the page size of the database
- the page size must be a power of two between 512 and 65536 inclusive
- DBOpen method will set the PageSize to 4096 (if the database is not encrypted), which sounds better than the default 1024 value - you should not have to set this property usually
- setting this property will only cause an immediate change in the page size if it is issued while the database is still empty, prior to the first CREATE TABLE statement; if this property is used to specify a new page size just prior to running the VACUUM command and if the database is not in WAL journal mode then VACUUM will change the page size to the new value for the newly created database file

property Password: RawUTF8 read fPassword;

Read-only access to the SQlite3 password used for encryption
- may be a JSON-serialized TSynSignerParams object, or will use AES-OFB-128 after SHAKE_128 with rounds=1000 and a fixed salt on plain password text

property SQLite3Library: TSQLite3Library read GetSQLite3Library;

The SQlite3 library which is currently running
- part of TSQLDatabase published properties, to publish e.g. Version

property Synchronous: TSQLSynchronousMode read GetSynchronous write SetSynchronous;

Query or change the SQLite3 file-based synchronization mode, i.e. the way it waits for the data to be flushed on hard drive
- default smFull is very slow, but achieve 100% ACID behavior
- smNormal is faster, and safe until a catastrophic hardware failure occurs
- smOff is the fastest, data should be safe if the application crashes, but database file may be corrupted in case of failure at the wrong time

property TransactionActive: boolean read fTransactionActive;

Return TRUE if a Transaction begun

property UseCache: boolean read GetUseCache write SetUseCache;

If this property is set, all ExecuteJSON() responses will be cached
- cache is flushed on any write access to the DB (any not SELECT statement)
- cache is consistent only if ExecuteJSON() Expand parameter is constant
- cache is used by TSQLDatabase.ExecuteJSON() and TSQLTableDB.Create()

property user_version: cardinal read GetUserVersion write SetUserVersion;

Retrieve or set the user_version stored in the SQLite3 database file
- user-version is a 32-bit signed integer stored in the database header
- it can be used to change the database in case of format upgrade (e.g. refresh some hand-made triggers)
property WALMode: Boolean read GetWALMode write SetWALMode;

Query or change the Write-Ahead Logging mode for the database
- beginning with version 3.7 of the SQLite3 engine, a new "Write-Ahead Log" option (hereafter referred to as "WAL") is optionally available
- WAL might be very slightly slower (perhaps 1% or 2% slower) than the traditional rollback-journal approach in applications that do mostly reads and seldom write; but WAL provides more concurrency as readers do not block writers and a writer does not block readers. Reading and writing can proceed concurrently. With our SQLite3 framework, it's not needed.
- by default, this option is not set: only implement if you really need it, but our SQLite3 framework use locked access to the database, so there should be no benefit of WAL for the framework; but if you call directly TSQLDatabase instances in your code, it may be useful to you

**TSQLBlobStream = class(TStream)**

*Used to read or write a BLOB Incrementaly*
- data is read/written directly from/to the SQLite3 BTree
- data can be written after a TSQLRequest.BindZero() call to reserve memory
- this TStream has a fixed size, but Position property can be used to rewind

*Used for DI-2.2.1 (page 2548).*

**constructor** Create(aDB: TSQLite3DB; const DBName, TableName, ColumnName: RawUTF8; RowID: Int64; ReadWrite: boolean);

*Opens a BLOB located in row RowID, column ColumnName, table TableName in database DBName; in other words, the same BLOB that would be selected by:*

```sql
SELECT ColumnName FROM DBName.TableName WHERE rowid = RowID;
```

**destructor** Destroy; override;

*Release the BLOB object*

**function** Read(var Buffer; Count: Longint): Longint; override;

*Read Count bytes from the opened BLOB in Buffer*

**function** Seek(Offset: Longint; Origin: Word): Longint; override;

*Change the current read position*

**function** Write(const Buffer; Count: Longint): Longint; override;

*Write is allowed for in-place replacement (resizing is not allowed)*

- Create() must have been called with ReadWrite=true

**procedure** ChangeRow(RowID: Int64);

*Reuse this class instance with another row of the same table*

- will update the stream size, and also rewind position to the beginning
- it is actually faster than creating a new TSQLBlobStream instance

**property** Handle: TSQLite3Blob read fBlob;

*Read-only access to the BLOB object handle*

**TSQLDatabaseBackupThread = class(TThread)**

*Background thread used for TSQLDatabase.BackupBackground() process*
constructor Create(Backup: TSQLite3Backup; Source, Dest: TSQLDatabase;
StepPageNumber, StepSleepMS: Integer; SynLzCompress: boolean; OnProgress:
TSQLDatabaseBackupEvent; OwnerDest: boolean=true); reintroduce;
Initialize the background thread
- execution is started immediately - caller may call the WaitFor inherited method to run the
process in blocking mode

property BackupDestFile: TFileName read fBackupDestFile;
The backup target database file name

property DestDB: TSQLDatabase read fDestDB;
The destination database of the backup process

property FailureError: Exception read fError;
The raised exception in case of backupFailure notification

property SourceDB: TSQLDatabase read fSourceDB;
The source database of the backup process

property Step: TSQLDatabaseBackupEventStep read fStep;
The current state of the backup process
- only set before a call to TSQLDatabaseBackupEvent

property StepNumberToFinish: integer read fStepNumberToFinish;
The number of pages which remain before end of backup
- only set before a call to TSQLDatabaseBackupEvent with backupStep* event

property StepNumberTotal: integer read fStepNumberTotal;
The number of pages for the whole database
- only set before a call to TSQLDatabaseBackupEvent with backupStep* event

property StepSynLzCompress: boolean read fStepSynLzCompress;
if .dbsynlz compression would be done on the backup file
- would use FileSynLZ(), so compress in chunks of 128 MB

Types implemented in the SynSQLite3 unit

TOnSQLStoredProc = procedure(const Statement: TSQLRequest) of object;
Stored Procedure prototype, used by TSQLDataBase.Execute() below
- called for every row of a Statement
- the implementation may update the database directly by using a local or shared TSQLRequest
- the TSQLRequest may be shared and prepared before the call for even faster access than with a
local TSQLRequest
- no TSQLDataBase or higher levels objects can be used inside this method, since all locking and
try..finally protection is outside it
- can optionally trigger a ESQLite3Exception on any error

TSQLAuthorizerCallback = function(pUserData: Pointer; code: Integer; const zTab, zCol,
zDb, zAuthContext: PAnsiChar): Integer; cdecl;
Compile-Time Authorization Callback prototype
- The authorizer callback is invoked as SQL statements are being compiled by sqlite3.prepare2() e.g.
- The authorizer callback should return SQLITE_OK to allow the action, SQLITE_IGNORE to disallow
the specific action but allow the SQL statement to continue to be compiled, or SQLITE_DENY to
cause the entire SQL statement to be rejected with an error.
- If the authorizer callback returns any value other than SQLITE_IGNORE, SQLITE_OK, or SQLITE_DENY then the sqlite3.prepare_v2() or equivalent call that triggered the authorizer will fail with an error message.
- The first pUserData parameter to the authorizer callback is a copy of the third parameter to the sqlite3.set_authorizer() interface.
- The second parameter to the callback is an integer action code that specifies the particular action to be authorized.
- The third through sixth parameters to the callback are zero-terminated strings that contain additional details about the action to be authorized.
- Here is a list of handled code constant, and their associated zTab / zCol parameters:

<table>
<thead>
<tr>
<th>const</th>
<th>zTab</th>
<th>zCol</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLITE_CREATE_INDEX</td>
<td>Index Name</td>
<td>Table Name</td>
</tr>
<tr>
<td>SQLITE_CREATE_TABLE</td>
<td>Table Name</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_CREATE_TEMP_INDEX</td>
<td>Index Name</td>
<td>Table Name</td>
</tr>
<tr>
<td>SQLITE_CREATE_TEMP_TABLE</td>
<td>Table Name</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_CREATE_TEMP_TRIGGER</td>
<td>Trigger Name</td>
<td>Table Name</td>
</tr>
<tr>
<td>SQLITE_CREATE_TEMP_VIEW</td>
<td>View Name</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_CREATE_TRIGGER</td>
<td>Trigger Name</td>
<td>Table Name</td>
</tr>
<tr>
<td>SQLITE_CREATE_VIEW</td>
<td>View Name</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_DELETE</td>
<td>Table Name</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_DROP_INDEX</td>
<td>Index Name</td>
<td>Table Name</td>
</tr>
<tr>
<td>SQLITE_DROP_TABLE</td>
<td>Table Name</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_DROP_TEMP_INDEX</td>
<td>Index Name</td>
<td>Table Name</td>
</tr>
<tr>
<td>SQLITE_DROP_TEMP_TABLE</td>
<td>Table Name</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_DROP_TEMP_TRIGGER</td>
<td>Trigger Name</td>
<td>Table Name</td>
</tr>
<tr>
<td>SQLITE_DROP_TEMP_VIEW</td>
<td>View Name</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_DROP_TRIGGER</td>
<td>Trigger Name</td>
<td>Table Name</td>
</tr>
<tr>
<td>SQLITE_DROP_VIEW</td>
<td>View Name</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_INSERT</td>
<td>Table Name</td>
<td>Column Name</td>
</tr>
<tr>
<td>SQLITE_PRAGMA</td>
<td>Pragma Name</td>
<td>1st arg or nil</td>
</tr>
<tr>
<td>SQLITE_READ</td>
<td>Table Name</td>
<td>Column Name</td>
</tr>
<tr>
<td>SQLITE_SELECT</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_TRANSACTION</td>
<td>Operation</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_UPDATE</td>
<td>Table Name</td>
<td>Column Name</td>
</tr>
<tr>
<td>SQLITE_ATTACH</td>
<td>Filename</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_DETACH</td>
<td>Database Name</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_ALTER_TABLE</td>
<td>Database Name</td>
<td>Table Name</td>
</tr>
<tr>
<td>SQLITE_REINDEX</td>
<td>Index Name</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_ANALYZE</td>
<td>Table Name</td>
<td>nil</td>
</tr>
<tr>
<td>SQLITE_CREATE_VTABLE</td>
<td>Table Name</td>
<td>Module Name</td>
</tr>
<tr>
<td>SQLITE_DROP_VTABLE</td>
<td>Table Name</td>
<td>Module Name</td>
</tr>
<tr>
<td>SQLITE_FUNCTION</td>
<td>nil</td>
<td>Function Name</td>
</tr>
<tr>
<td>SQLITE_SAVEPOINT</td>
<td>Operation</td>
<td>Savepoint Name</td>
</tr>
</tbody>
</table>

- The 5th parameter to the authorizer callback is the name of the database ('main', 'temp', etc.) if applicable.
- The 6th parameter to the authorizer callback is the name of the inner-most trigger or view that is responsible for the access attempt or nil if this access attempt is directly from top-level SQL code.

TSQLE_BUSYHandler = function(user: pointer; count: integer): integer; cdecl;

SQLite3 callback prototype to handle SQLITE_BUSY errors
- The first argument to the busy handler is a copy of the user pointer which is the third argument to sqlite3.busy_handler().
- The second argument to the busy handler callback is the number of times that the busy handler has been invoked for this locking event.
- If the busy callback returns 0, then no additional attempts are made to access the database and SQLITE_BUSY or SQLITE_IOERR_BLOCKED is returned.
- If the callback returns non-zero, then another attempt is made to open the database for reading and the cycle repeats.
TSQLCollateFunc = function(CollateParam: pointer; s1Len: integer; s1: pointer; s2Len: integer; s2: pointer) : integer; cdecl;

SQLite3 collation (i.e. sort and comparison) function prototype
- this function MUST use s1Len and s2Len parameters during the comparison: s1 and s2 are not zero-terminated
- used by sqlite3.create_collation low-level function

TSQLCommitCallback = function(pArg: Pointer): Integer; cdecl;

Commit And Rollback Notification Callback function after sqlite3.commit_hook() or sqlite3.rollback_hook() registration
- The callback implementation must not do anything that will modify the database connection that invoked the callback. Any actions to modify the database connection must be deferred until after the completion of the sqlite3.step() call that triggered the commit or rollback hook in the first place. Note that sqlite3.prepare_v2() and sqlite3.step() both modify their database connections for the meaning of "modify" in this paragraph.
- When the commit hook callback routine returns zero, the COMMIT operation is allowed to continue normally. If the commit hook returns non-zero, then the COMMIT is converted into a ROLLBACK. The rollback hook is invoked on a rollback that results from a commit hook returning non-zero, just as it would be with any other rollback.
- For the purposes of this API, a transaction is said to have been rolled back if an explicit "ROLLBACK" statement is executed, or an error or constraint causes an implicit rollback to occur. The rollback callback is not invoked if a transaction is automatically rolled back because the database connection is closed.

TSQLDatabaseBackupEvent = function(Sender: TSQLDatabaseBackupThread): boolean of object;

Callback called asynchronously during TSQLDatabase.BackupBackground()
- implementation should return TRUE to continue the process: if the method returns FALSE, backup will be aborted, and destination file deleted
- this method allows to monitor the backup process, thanks to TSQLDatabaseBackupThread properties (especially the Step property)
- this method will be executed in the context of the associated TSQLDatabaseBackupThread: so you should use Synchronize() to update the UI

TSQLDatabaseBackupEventStep = ( backupNone, backupStart, backupSuccess, backupFailure, backupStepOk, backupStepBusy, backupStepLocked, backupStepSynLz );

Kind of event triggered during TSQLDatabase.BackupBackground() process
- you can use (Sender.Step in backupAnyStep), to check for normal step, or (Sender.Step in backupFinished) to check for process end

TSQLDataBaseTransactionBehaviour = ( tbDeferred, tbImmediate, tbExclusive );

TSQLDatabase.TransactionBegin can be deferred, immediate, or exclusive
- tbDeferred means that no locks are acquired on the database until the database is first accessed. Thus with a deferred transaction, the BEGIN statement itself does nothing to the filesystem. Locks are not acquired until the first read or write operation. The first read operation against a database creates a SHARED lock and the first write operation creates a RESERVED lock. Because the acquisition of locks is deferred until they are needed, it is possible that another thread or process could create a separate transaction and write to the database after the BEGIN on the current thread has executed.
- If the transaction is tbImmediate, then RESERVED locks are acquired on all databases as soon as the BEGIN command is executed, without waiting for the database to be used. After a BEGIN IMMEDIATE, no other database connection will be able to write to the database or do a BEGIN
IMMEDIATE or BEGIN EXCLUSIVE. Other processes can continue to read from the database, however.
- A dbExclusive transaction causes EXCLUSIVE locks to be acquired on all databases. After a BEGIN EXCLUSIVE, no other database connection except for read_uncommitted connections will be able to read the database and no other connection without exception will be able to write the database until the transaction is complete.

```pascal
TSQLDestroyPtr = procedure(p: pointer); cdecl;
Type for a custom destructor for the text or BLOB content
- set to @sqlite3InternalFree if a Value must be released via Freemem()
- set to @sqlite3InternalFreeObject if a Value must be released via TObject(p).Free

TSQLFunctionFinal = procedure(Context: TSQLite3FunctionContext); cdecl;
SQLite3 user final aggregate callback prototype

TSQLFunctionFunc = procedure(Context: TSQLite3FunctionContext; argc: integer; var argv: TSQLite3ValueArray); cdecl;
SQLite3 user function or aggregate callback prototype
- argc is the number of supplied parameters, which are available in argv[] (you can call ErrorWrongNumberOfArgs(Context) in case of unexpected number)
- use sqlite3.value_*(argv[*]) functions to retrieve a parameter value
- then set the result using sqlite3.result_*(Context,*) functions

TSQLite3Backup = type PtrUInt;
  Internaly store a SQLite3 Backup process handle

TSQLite3Blob = type PtrUInt;
  Internaly store the SQLite3 blob handle
  Used for DI-2.2.1 (page 2548).

TSQLite3DB = type PtrUInt;
  Internaly store the SQLite3 database handle
  Used for DI-2.2.1 (page 2548).

TSQLite3ErrorCode = (secUnknown, secOK, secERROR, secINTERNAL, secPERM, secABORT, secBUSY, secLOCKED, secNOMEM, secREADONLY, secINTERRUPT, secIOERR, secCORRUPT, secNOTFOUND, secFULL, secCANTOPEN, secPROTOCOL, secEMPTY, secSCHEMA, secTOOBIG, secCONTRAINT, secMISMATCH, secMISUSE, secNOLFS, secAUTH, secFORMAT, secRANGE, secNOTADB, secROW, secDONE);
  The main possible return codes, including error codes

TSQLite3FunctionContext = type PtrUInt;
  Internaly store a SQLite3 Function Context Object
  - The context in which an SQL function executes is stored in an sqlite3.context object, which is mapped to this TSQLite3FunctionContext type
  - A pointer to an sqlite3.context object is always first parameter to application-defined SQL functions, i.e. a TSQLFunctionFunc prototype

TSQLite3Statement = type PtrUInt;
  Internaly store the SQLite3 statement handle
  - This object is variously known as a "prepared statement" or a "compiled SQL statement" or simply as a "statement".
- Create the object using sqlite3.prepare_v2() or a related function.
- Bind values to host parameters using the sqlite3.bind_*() interfaces.
- Run the SQL by calling sqlite3.step() one or more times.
- Reset the statement using sqlite3.reset() then go back to "Bind" step. Do this zero or more times.
- Destroy the object using sqlite3.finalize().

*Used for DI-2.2.1 (page 2548).*

**TSQLite3Value = type** PtrUInt;

*Internally store a SQLite3 Dynamically Typed Value Object*
- SQLite uses the sqlite3.value object to represent all values that can be stored in a database table, which are mapped to this TSQLite3Value type
- SQLite uses dynamic typing for the values it stores
- Values stored in sqlite3.value objects can be integers, floating point values, strings, BLOBs, or NULL

*Used for DI-2.2.1 (page 2548).*

**TSQLite3ValueArray = array[0..63] of TSQLite3Value;**

*Internally store of SQLite3 values, as used by TSQLFunctionFunc*

*Used for DI-2.2.1 (page 2548).*

**TSQLLimitCategory = ( lcLength, lcSQLLength, lcColumn, lcExprDepth, lcCompoundSelect, lcVDBEop, lcFunctionArg, lcAttached, lcLikePatternLength, lcVariableNumber, lcTriggerDepth );**

*Available Run-Time limit categories*
- lcLength The maximum size of any string or BLOB or table row, in bytes.
- lcSQLLength The maximum length of an SQL statement, in bytes.
- lcColumn The maximum number of columns in a table definition or in the result set of a SELECT or the maximum number of columns in an index or in an ORDER BY or GROUP BY clause.
- lcExprDepth The maximum depth of the parse tree on any expression.
- lcCompoundSelect The maximum number of terms in a compound SELECT statement.
- lcVDBEop The maximum number of instructions in a virtual machine program used to implement an SQL statement. This limit is not currently enforced, though that might be added in some future release of SQLite.
- lcFunctionArg The maximum number of arguments on a function.
- lcAttached The maximum number of attached databases.
- lcLikePatternLength The maximum length of the pattern argument to the LIKE or GLOB operators.
- lcVariableNumber The maximum number of parameters in an SQL statement.
- lcTriggerDepth The maximum depth of recursion for triggers.

**TSQLLockingMode = ( lmNormal, lmExclusive );**

*Available file-level database connection locking-mode*
- lmNormal locking-mode (the default unless overridden at compile-time using SQLITE_DEFAULT_LOCKING_MODE), a database connection unlocks the database file at the conclusion of each read or write transaction.
- when the locking-mode is set to lmExclusive, the database connection never releases file-locks. The first time the database is read in lmExclusive mode, a shared lock is obtained and held. The first time the database is written, an exclusive lock is obtained and held. Database locks obtained by a connection in lmExclusive mode may be released either by closing the database connection, or by setting the locking-mode back to lmNormal using this pragma and then accessing the database file (for read or write). Simply setting the locking-mode to lmNormal is not enough - locks are not
released until the next time the database file is accessed.
- ImExclusive gives much better write performance, and could be used when needed, in case of a heavy loaded mORMot server

```pascal
TSQLProfileCallback = procedure(ProfileArg: Pointer; Profile: PUTF8Char; ProfileNanoSeconds: Int64); cdecl;
Callback function registered by sqlite3.profile()
- this procedure will be invoked as each SQL statement finishes
- warning: sqlite3.profile() function is considered experimental and is subject to change in future versions of SQLite

TSQLStatementCacheDynArray = array of TSQLStatementCache;
Used to store all prepared statement

TSQLSynchronousMode = ( smOff, smNormal, smFull );
Available file-level write access wait mode of the SQLite3 engine
- when synchronous is smFull (which is the default setting), the SQLite database engine will use the xSync method of the VFS to ensure that all content is safely written to the disk surface prior to continuing. This ensures that an operating system crash or power failure will not corrupt the database. FULL synchronous is very safe, but it is also slower.
- when synchronous is smNormal, the SQLite database engine will still sync at the most critical moments, but less often than in FULL mode. There is a very small (though non-zero) chance that a power failure at just the wrong time could corrupt the database in NORMAL mode. But in practice, you are more likely to suffer a catastrophic disk failure or some other unrecoverable hardware fault.
- when synchronous is smOff, SQLite continues without syncing as soon as it has handed data off to the operating system. If the application running SQLite crashes, the data will be safe, but the database might become corrupted if the operating system crashes or the computer loses power before that data has been written to the disk surface. On the other hand, some operations are as much as 50 or more times faster with synchronous OFF.

```

TSQLTraceCallback = procedure(Trace: TSQLTraceMask; UserData,P,X: pointer); cdecl;
Callback function registered by sqlite3.trace_v2()
- the Trace argument has one of the TSQLTraceMask items set, to indicate why the callback was invoked
- UserData argument is a copy of the context pointer, as provided at sqlite3.trace_v2() call
- P and X arguments are pointers whose meanings depend on Trace content: see TSQLTraceMask for the various use cases

TSQLTraceMask = set of (stmStmt, stmProfile, stmRow, stmClose);
Events monitored by sqlite3.trace_v2() tracing logic
- stmStmt callback is invoked when a prepared statement first begins running and possibly at other times during the execution of the prepared statement, such as at the start of each trigger subprogram. The P argument is a pointer to the prepared statement. The X argument is a pointer to a string which is the unexpanded SQL text of the prepared statement or an SQL comment that indicates the invocation of a trigger.
- stmProfile callback provides approximately the same information as was provided by the deprecated sqlite3.profile() callback. The P argument is a pointer to the prepared statement and the X argument points to a 64-bit integer which is the estimated of the number of nanosecond that the prepared statement took to run. The stmProfile callback is invoked when the statement finishes.
- stmRow callback is invoked whenever a prepared statement generates a single row of result. The P argument is a pointer to the prepared statement and the X argument is unused.
- stmClose callback is invoked when a database connection closes. The P argument is a pointer to the
database connection object and the X argument is unused.

```delphi
TSQLUpdateCallback = procedure(pUpdateArg: Pointer; op: Integer; const zDb, zTbl: PUTF8Char; iRowID: Int64); cdecl;
```

*Callback function invoked when a row is updated, inserted or deleted, after sqlite3.update_hook() registration*
- The first pUpdateArg argument is a copy of the third argument to sqlite3.update_hook().
- The second op argument is one of SQLITE_INSERT, SQLITE_DELETE, or SQLITE_UPDATE, depending on the operation that caused the callback to be invoked.
- The third and fourth zDB / zTbl arguments contain pointers to the database and table name containing the affected row.
- The final iRowID parameter is the rowid of the row. In the case of an update, this is the rowid after the update takes place.
- The update hook implementation must not do anything that will modify the database connection that invoked the update hook. Any actions to modify the database connection must be deferred until after the completion of the sqlite3.step() call that triggered the update hook. Note that sqlite3.prepare_v2() and sqlite3.step() both modify their database connections for the meaning of “modify” in this paragraph.

**Constants implemented in the SynSQLite3 unit**

```delphi
backupAnyStep = [backupStepOk, backupStepBusy, backupStepLocked, backupStepSynLz];
```

*Identify the iterative step events during TSQLDatabase.BackupBackground()*
- you can use (Sender.Step in backupAnyStep), to check for normal step

```delphi
backupFinished = [backupSuccess, backupFailure];
```

*Identify the end step events during TSQLDatabase.BackupBackground()*
- you can use (Sender.Step in backupFinished) to check for process end

```delphi
SQLDATABASE_NOCACHE: RawUTF8 = '/*nocache*/';
```

*A magic text constant which will prevent any JSON result to be cached in TSQLDatabase, if present in the SQL statement*
- to be used e.g. when you put some pointers as bound parameters

```delphi
SQLITE3_MAGIC = $ABA5A5AB;
```

*The "magic" number used to identify dbsynlz compressed files, as created by TSQLDatabase.BackupSynLZ() or if SynLZCompress parameter is TRUE for the TSQLDatabase.BackupBackground() method*
- note that the SynDBExplorer tool is able to recognize such files, and open them directly or use the DBSynLZ.dpr command-line sample tool

```delphi
SQLITE_ABORT = 4;
```

*Sqlite_exec() return code: Callback routine requested an abort*

```delphi
SQLITE_ANY = 5;
```

*Sqlite3.create_function don't care about text encoding*

```delphi
SQLITE_AUTH = 23;
```

*Sqlite_exec() return code: Authorization denied*

```delphi
SQLITE_BLOB = 4;
```

*Internal SQLite3 type as Blob*

```delphi
SQLITE_BUSY = 5;
```
**Sqlite_exec() return code: The database file is locked**

`SQLITE_CANTOPEN = 14;`

**Sqlite_exec() return code: Unable to open the database file**

`SQLITE_CONSTRAINT = 19;`

**Sqlite_exec() return code: Abort due to constraint violation**

`SQLITE_CORRUPT = 11;`

**Sqlite_exec() return code: The database disk image is malformed**

`SQLITE_DONE = 101;`

**Sqlite3.step() return code: has finished executing**

`SQLITE_EMPTY = 16;`

**Sqlite_exec() return code: Database is empty**

`SQLITE_ERROR = 1;`

**Sqlite_exec() return code: SQL error or missing database - legacy generic code**

`SQLITE_ERRORS = [SQLITE_ERROR..SQLITE_ROW];`

Possible error codes for `sqlite_exec()` and `sqlite3.step()` - as verified by `sqlite3_check()`

```
SQLITE_FILE_HEADER: array[0..15] of AnsiChar = 'SQLite format 3';

The "magic" 16 bytes header stored at the beginning of every SQLite3 file
```

`SQLITE_FLOAT = 2;`

**Internal SQLite type as Floating point value**

`SQLITE_FORMAT = 24;`

**Sqlite_exec() return code: Auxiliary database format error**

`SQLITE_FULL = 13;`

**Sqlite_exec() return code: Insertion failed because database is full**

`SQLITE_INTEGER = 1;`

**Internal SQLite type as Integer**

`SQLITE_INTERNAL = 2;`

**Sqlite_exec() return code: An internal logic error in SQLite**

`SQLITE_INTERRUPT = 9;`

**Sqlite_exec() return code: Operation terminated by sqlite3.interrupt()**

`SQLITE_IOERR = 10;`

**Sqlite_exec() return code: Some kind of disk I/O error occurred**

`SQLITE_LOCKED = 6;`

**Sqlite_exec() return code: A table in the database is locked**

`SQLITE_MEMORY_DATABASE_NAME = ':memory:';`

**Pseudo database file name used to create an in-memory database**

- an SQLite database is normally stored in a single ordinary disk file - however, in certain circumstances, the database might be stored in memory, if you pass `SQLITE_MEMORY_DATABASE_NAME` to TSQLDatabase.Create() instead of a real disk file name
- this instance will cease to exist as soon as the database connection is closed, i.e. when calling TSQLDatabase.Free
- every ':memory:' database is distinct from every other - so, creating two TSQLDatabase instances each with the filename SQLITE_MEMORY_DATABASE_NAME will create two independent in-memory databases

```
SQLITE_MISMATCH = 20;
Sqlite_exec() return code: Data type mismatch

SQLITE_MISUSE = 21;
Sqlite_exec() return code: Library used incorrectly

SQLITE_NOLFS = 22;
Sqlite_exec() return code: Uses OS features not supported on host

SQLITE_NOMEM = 7;
Sqlite_exec() return code: A malloc() failed

SQLITE_NOTADB = 26;
Sqlite_exec() return code: File opened that is not a database file

SQLITE_NOTFOUND = 12;
Sqlite_exec() return code: (Internal Only) Table or record not found

SQLITE_NULL = 5;
Internal SQLite3 type as NULL

SQLITE_OK = 0;
Sqlite_exec() return code: no error occurred

SQLITE_OPEN_CREATE = $00000004;
In conjunction with SQLITE_OPEN_READWRITE, optionally create the database file if it does not exist
- The database is opened for reading and writing if possible, or reading only if the file is write protected by the operating system
- In either case the database must already exist, otherwise an error is returned

SQLITE_OPEN_FULLMUTEX = $00010000;
Ok for sqlite3_open_v2() If the SQLITE_OPEN_FULLMUTEX flag is set then the database connection opens in the serialized threading mode unless single-thread was previously selected at compile-time or start-time
- Ok for sqlite3.open_v2(), in conjunction with SQLITE_OPEN_READONLY, SQLITE_OPEN_READWRITE, (SQLITE_OPEN_READWRITE or SQLITE_OPEN_CREATE)

SQLITE_OPEN_MEMORY = $00000800;
Ok for sqlite3_open_v2() If the SQLITE_OPEN_NOMUTEX flag is set, then the database will remain in memory
- Ok for sqlite3.open_v2(), in conjunction with SQLITE_OPEN_READONLY, SQLITE_OPEN_READWRITE, (SQLITE_OPEN_READWRITE or SQLITE_OPEN_CREATE)

SQLITE_OPEN_NOMUTEX = $00008000;
Ok for sqlite3_open_v2() If the SQLITE_OPEN_NOMUTEX flag is set, then the database connection opens in the multi-thread threading mode as long as the single-thread mode has not been set at compile-time or start-time
- Ok for sqlite3.open_v2(), in conjunction with SQLITE_OPEN_READONLY, SQLITE_OPEN_READWRITE, (SQLITE_OPEN_READWRITE or SQLITE_OPEN_CREATE)
SQLite_OPEN_PRIVATECACHE = $00040000;
Ok for sqlite3_open_v2() The SQLite_OPEN_PRIVATECACHE flag causes the database connection to
not participate in shared cache mode even if it is enabled
- Ok for sqlite3_open_v2(), in conjunction with SQLite_OPEN_READONLY, SQLite_OPEN_READWRITE,
(SQLITE_OPEN_READWRITE or SQLite_OPEN_CREATE)

SQLite_OPEN_READONLY = $00000001;
The database is opened in read-only mode
- if the database does not already exist, an error is returned
- Ok for sqlite3_open_v2()

SQLite_OPEN_READWRITE = $00000002;
The database is opened for reading and writing if possible, or reading only if the file is write
protected by the operating system
- In either case the database must already exist, otherwise an error is returned
- Ok for sqlite3_open_v2()

SQLite_OPEN_SHAREDCACHE = $00020000;
Ok for sqlite3_open_v2() The SQLite_OPEN_SHAREDCACHE flag causes the database connection to
be eligible to use shared cache mode, regardless of whether or not shared cache is enabled using
sqlite3.enable_shared_cache()
- Ok for sqlite3_open_v2(), in conjunction with SQLite_OPEN_READONLY, SQLite_OPEN_READWRITE,
(SQLITE_OPEN_READWRITE or SQLite_OPEN_CREATE)

SQLite_OPEN_URI = $00000040;
URI filename interpretation is enabled if the SQLite_OPEN_URI flag is set in the fourth argument to
sqlite3_open_v2(), or if it has been enabled globally using the SQLite_CONFIG_URI option with the
sqlite3.config() method or by the SQLite_USE_URI compile-time option.
- As of SQLite version 3.7.7, URI filename interpretation is turned off by default, but future releases
of SQLite might enable URI filename interpretation by default
- Ok for sqlite3_open_v2(), in conjunction with SQLite_OPEN_READONLY, SQLite_OPEN_READWRITE,
(SQLITE_OPEN_READWRITE or SQLite_OPEN_CREATE)

SQLite_PERM = 3;
Sqlite_exec() return code: Access permission denied

SQLite_PROTOCOL = 15;
Sqlite_exec() return code: (Internal Only) Database lock protocol error

SQLite_RANGE = 25;
Sqlite_exec() return code: 2nd parameter to sqlite3.bind out of range

SQLite_READONLY = 8;
Sqlite_exec() return code: Attempt to write a read-only database

SQLite_ROW = 100;
Sqlite3.step() return code: another result row is ready

SQLite_SCHEMA = 17;
Sqlite_exec() return code: The database schema changed, and unable to be recompiled

SQLite_STATIC = pointer(0);
DestroyPtr set to SQLite_STATIC if data is constant and will never change
- SQLite assumes that the text or BLOB result is in constant space and does not copy the content of
the parameter nor call a destructor on the content when it has finished using that result

```pascal
SQLITE_TEXT = 3;

Internal SQLite3 type as Text
```

```pascal
SQLITE_TOOBIG = 18;

sqlite_exec() return code: Too much data for one row of a table
```

```pascal
SQLITE_TRANSIENT = pointer(-1);

DestroyPtr set to SQLITE_TRANSIENT for SQLite3 to make a private copy of the data into space obtained from from sqlite3.malloc() before it returns
- this is the default behavior in our framework
- note that we discovered that under Win64, sqlite3.result_text() expects SQLITE_TRANSIENT_VIRTUALTABLE=pointer(integer(-1)) and not pointer(-1)
```

```pascal
SQLITE_TRANSIENT_VIRTUALTABLE = pointer(integer(-1));

DestroyPtr set to SQLITE_TRANSIENT_VIRTUALTABLE for setting results to SQLite3 virtual tables columns
- due to a bug of the SQLite3 engine under Win64
```

```pascal
SQLITE_UTF16 = 4;

Text is UTF-16 encoded, using the system native byte order
```

```pascal
SQLITE_UTF16BE = 3;

Text is UTF-16 BE encoded
```

```pascal
SQLITE_UTF16LE = 2;

Text is UTF-16 LE encoded
```

```pascal
SQLITE_UTF16_ALIGNED = 8;

Used by sqlite3.create_collation() only
```

```pascal
SQLITE_UTF8 = 1;

Text is UTF-8 encoded
```

```pascal
SQL_GET_TABLE_NAMES = 'SELECT name FROM sqlite_master WHERE type=''table'' AND name NOT LIKE ''sqlite_%'';';

SQL statement to get all tables names in the current database file (taken from official SQLite3 documentation)
```

### Functions or procedures implemented in the SynSQLite3 unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckNumberOfArgs</td>
<td>Wrapper around sqlite3.result_error() validating the expected number of arguments</td>
<td>1707</td>
</tr>
<tr>
<td>ErrorCodeToText</td>
<td>Convert a TSQLite3ErrorCode item into the corresponding SQLite constant name</td>
<td>1707</td>
</tr>
<tr>
<td>ErrorWrongNumberOfArgs</td>
<td>Wrapper around sqlite3.result_error() to be called if wrong number of arguments</td>
<td>1707</td>
</tr>
<tr>
<td>ExceptionToSqlite3Err</td>
<td>Create a TSQLite3Module.pzErr UTF-8 text buffer according to the given Delphi exception</td>
<td>1707</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>IsSQLite3File</td>
<td>Check from the file beginning if sounds like a valid SQLite3 file</td>
<td>1707</td>
</tr>
<tr>
<td>IsSQLite3FileEncrypted</td>
<td>Check if sounds like an encrypted SQLite3 file</td>
<td>1708</td>
</tr>
<tr>
<td>JsonToSQLite3Context</td>
<td>Set a JSON value into a SQLite3 result context</td>
<td>1708</td>
</tr>
<tr>
<td>RawUTF8ToSQLite3Context</td>
<td>Set a UTF-8 string into a SQLite3 result context</td>
<td>1708</td>
</tr>
<tr>
<td>sqlite3InternalFree</td>
<td>An internal function which calls Freemem(p)</td>
<td>1708</td>
</tr>
<tr>
<td>sqlite3InternalFreeObject</td>
<td>An internal function which calls TObject(p).Free</td>
<td>1708</td>
</tr>
<tr>
<td>sqlite3InternalFreeRawByteString</td>
<td>An internal function which calls RawByteString(p) := ''</td>
<td>1708</td>
</tr>
<tr>
<td>SQLite3ValueToSQLVar</td>
<td>Set a SQLite3 value into a TSQLVar</td>
<td>1708</td>
</tr>
<tr>
<td>sqlite3_check</td>
<td>Test the result state of a sqlite3.*() function</td>
<td>1708</td>
</tr>
<tr>
<td>sqlite3_resultToErrorCode</td>
<td>Convert a SQLite3 result code into a TSQLite3ErrorCode item</td>
<td>1708</td>
</tr>
<tr>
<td>sqlite3_resultToErrorText</td>
<td>Convert a SQLite3 result code into the corresponding SQLite constant name</td>
<td>1708</td>
</tr>
<tr>
<td>SqlQueryToSQLite3Context</td>
<td>Set a TSQLVar into a SQLite3 result context</td>
<td>1708</td>
</tr>
<tr>
<td>StatementCacheTotalTimeCompare</td>
<td>Comparison function using TSQLStatementCache.Timer.TimeInMicroSec</td>
<td>1709</td>
</tr>
<tr>
<td>VariantToSQLite3Context</td>
<td>Set a variant value into a SQLite3 result context</td>
<td>1709</td>
</tr>
</tbody>
</table>

**function** CheckNumberOfArgs(Context: TSQLite3FunctionContext; expected, sent: integer): boolean;
Wrapper around sqlite3.result_error() validating the expected number of arguments

**function** ErrorCodeToText(err: TSQLite3ErrorCode): RawUTF8;
Convert a TSQLite3ErrorCode item into the corresponding SQLite constant name
- e.g. ErrorCodeToText(secOK)='SQLITE_OK'

**procedure** ErrorWrongNumberOfArgs(Context: TSQLite3FunctionContext);
Wrapper around sqlite3.result_error() to be called if wrong number of arguments

**procedure** ExceptionToSqlite3Err(E: Exception; var pzErr: PUTF8Char);
Create a TSQLite3Module.pzErr UTF-8 text buffer according to the given Delphi exception

**function** IsSQLite3File(const FileName: TFileName; PageSize: PInteger=nil): boolean;
Check from the file beginning if sounds like a valid SQLite3 file
- returns true if a database file is encrypted or not
- optional retrieve the file page size from header
function IsSQLite3FileEncrypted(const FileName: TFileName): boolean;
Check if sounds like an encrypted SQLite3 file

procedure JsonToSQLite3Context(json: PUTF8Char; Context: TSQLite3FunctionContext);
Set a JSON value into a SQLite3 result context
- a JSON object or array would be returned at plain TEXT, or other simple JSON text or number would be returned as the corresponding SQLite3 value

procedure RawUTF8ToSQLite3Context(const Text: RawUTF8; Context: TSQLite3FunctionContext; VoidTextAsNull: boolean);
Set a UTF-8 string into a SQLite3 result context
- this function will use copy-on-write assignment of Text, with no memory allocation, then let sqlite3InternalFreeRawByteString release its reference count

procedure sqlite3InternalFree(p: pointer); cdecl;
An internal function which calls Freemem(p)
- can be used to free some PUTF8Char pointer allocated by Delphi Getmem()

procedure sqlite3InternalFreeObject(p: pointer); cdecl;
An internal function which calls TObject(p).Free
- can be used to free some Delphi class instance

procedure sqlite3InternalFreeRawByteString(p: pointer); cdecl;
An internal function which calls RawByteString(p) :="
- can be used to free some Delphi class instance
- use a local tmp: pointer variable to prepare the reference count, e.g.
  tmp := nil;
  RawUTF8(tmp) := Text; // fast COW assignment
  sqlite3.result_text(Context,tmp,length(Text)+1,sqlite3InternalFreeRawByteString);

procedure SQLite3ValueToSQLVar(Value: TSQLite3Value; var Res: TSQLVar);
Set a SQLite3 value into a TSQLVar
- will call the corresponding sqlite3.value_*() function to retrieve the data with the less overhead (e.g. memory allocation or copy) as possible

function sqlite3_check(DB: TSQLite3DB; aResult: integer; const SQL: RawUTF8=''): integer;
Test the result state of a sqlite3.*() function
- raise a ESQLite3Exception if the result state is within SQLITE_ERRORS
- return the result state otherwise (SQLITE_OK, SQLITE_ROW, SQLITE_DONE e.g.)

function sqlite3_resultToErrorCode(aResult: integer): TSQLite3ErrorCode;
Convert a SQLite3 result code into a TSQLite3ErrorCode item

function sqlite3_resultToErrorText(aResult: integer): RawUTF8;
Convert a SQLite3 result code into the corresponding SQLite constant name
- e.g. sqlite3_resultToErrorText(SQLITE_OK)='SQLITE_OK'

function SQLVarToSQLite3Context(const Res: TSQLVar; Context: TSQLite3FunctionContext): boolean;
Set a TSQLVar into a SQLite3 result context
- will call the corresponding sqlite3.result_*() function and return true, or will return false if the TSQLVar type is not handled
function StatementCacheTotalTimeCompare(const A,B): integer;

Comparison function using TSQLStatementCache.Timer.TimeInMicroSec

procedure VariantToSQLite3Context(const Value: Variant; Context: TSQLite3FunctionContext);

Set a variant value into a SQLite3 result context
- will call the corresponding sql3te3.result_*() function, using SQLVarToSQLite3Context() after a call to VariantToSQLVar()

Variables implemented in the SynSQLite3 unit

sqlite3: TSQLite3Library;

Global access to linked SQLite3 library API calls
- you should call sqlite3.open() instead of sqlite3_open() for instance
- points either to the statically linked sqlite3.obj, or to an external library (e.g. sqlite3.dll under Windows)
- your project should use EITHER SynSQLite3Static unit OR create a TSQLite3LibraryDynamic instance:
  FreeAndNil(sqlite3); // release any previous instance
  sqlite3 := TSQLite3LibraryDynamic.Create;
- caller should free the sqlite3 instance only with
  FreeAndNil(sqlite3);

to avoid issues with the automatic freeing in finalization section

SynSQLite3Log: TSynLogClass = TSynLog;

The TSynLog class used for logging for all our SynSQLite3 related functions
- you may override it with TSQLLog, if available from mORMot.pas
- since not all exceptions are handled specifically by this unit, you may better use a common TSynLog class for the whole application or module
27.36. SynSQLite3RegEx.pas unit

Purpose: REGEXP function for SQLite3 Database using PCRE library
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the SynSQLite3RegEx unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynSQLite3</td>
<td>SQLite3 Database engine direct access</td>
<td>1646</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

Functions or procedures implemented in the SynSQLite3RegEx unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateRegExpFunction</td>
<td>Register the REGEXP SQL function to a given SQLite3 engine instance</td>
<td>1710</td>
</tr>
</tbody>
</table>

```pascal
function CreateRegExpFunction(DB: TSQLite3DB): boolean;

Register the REGEXP SQL function to a given SQLite3 engine instance
- allow execution of statements as such:
  SELECT column FROM table WHERE column REGEXP '<here goes your expression>';```


27.37. *SynSQLite3Static*.pas unit

**Purpose**: SQLite3 3.33.0 Database engine - statically linked for Windows/Linux
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The *SynSQLite3Static* unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.2.1</td>
<td>The SQLite3 engine shall be embedded to the framework</td>
<td>2548</td>
</tr>
</tbody>
</table>

Units used in the *SynSQLite3Static* unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>SynCommons</em></td>
<td>Common functions used by most Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
<tr>
<td><em>SynCrypto</em></td>
<td>Fast cryptographic routines (hashing and cypher) - implements AES,XOR,ADLER32,MDS,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT - optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support) - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1139</td>
</tr>
<tr>
<td><em>SynSQLite3</em></td>
<td>SQLite3 Database engine direct access - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1646</td>
</tr>
</tbody>
</table>

**Objects implemented in the *SynSQLite3Static* unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLite3LibraryStatic</td>
<td>Access class to the static .obj SQLite3 engine</td>
<td>1711</td>
</tr>
</tbody>
</table>

**TSQLite3LibraryStatic** = class(TSQLite3Library)

*Access class to the static .obj SQLite3 engine*
- the initialization section of this unit calls:
  
```pascal
sqlite3 := TSQLite3LibraryStatic.Create;
```

therefore, adding *SynSQLite3Static* to your uses clause is enough to use the statically linked SQLite3 engine with *SynSQLite3*

*Used for DI-2.2.1 (page 2548).*
**constructor** Create; **override**;

*Fill the internal API reference s with the static .obj engine*

**destructor** Destroy; **override**;

*Unload the static library*

### Functions or procedures implemented in the *SynSQLite3Static* unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChangeSQLEncryptTablePassWord</td>
<td>Use this procedure to change the password for an existing SQLite3 database file</td>
<td>1712</td>
</tr>
<tr>
<td>IsOldSQLEncryptTable</td>
<td>Could be used to detect a database in old/deprecated/unsupported format (&lt;1.18.4413)</td>
<td>1712</td>
</tr>
<tr>
<td>OldSQLEncryptTablePasswordToPlain</td>
<td>This function may be used to create a plain database file from an existing one encrypted with our old/deprecated/unsupported format (&lt;1.18.4413)</td>
<td>1713</td>
</tr>
</tbody>
</table>

**function** ChangeSQLEncryptTablePassword(const FileName: TFileName; const OldPassWord, NewPassword: RawUTF8): boolean;

*Use this procedure to change the password for an existing SQLite3 database file*
- convenient and faster alternative to the sqlite3.rekey() API call
- conversion is done in-place at file level, with no SQL nor BTree pages involved, therefore it can process very big files with best possible speed
- the `OldPassWord` must be correct, otherwise the resulting file will be corrupted
- any password can be "" to mark no encryption as input or output
- the password may be a JSON-serialized TSynSignerParams object, or will use AES-OFB-128 after SHAKE_128 with rounds=1000 and a fixed salt on plain password text
- please note that this encryption is compatible only with SQLite3 files made with SynSQLiteStatic.pas unit (not external/official/wxsqlite3 dll)
- implementation is NOT compatible with the official SQLite Encryption Extension (SEE) file format, not the wxsqlite3 extension, but is (much) faster thanks to our SynCrypto AES-NI enabled unit
- if the key is not correct, a ESQLite3Exception will be raised with 'database disk image is malformed' (SQLITE_CORRUPT) at database opening
- see also IsSQLite3File/IsSQLite3FileEncrypted functions
- warning: this encryption is NOT compatible with our previous (<1.18.4413) cyphered format, which was much less safe (simple XOR on fixed tables), and was not working on any database size, making unclean patches to the official sqlite3.c amalgamation file, so is deprecated and unsupported any longer - see OldSQLEncryptTablePasswordToPlain() to convert your existing databases

**function** IsOldSQLEncryptTablePassWordToPlain(const FileName: TFileName): boolean;

*Could be used to detect a database in old/deprecated/unsupported format (<1.18.4413)*
- to call OldSQLEncryptTablePasswordToPlain + ChangeSQLEncryptTablePassWord and switch to the new format
procedure OldSQLEncryptTablePasswordToPlain(const FileName: TFileName; const OldPassword: RawUTF8);

This function may be used to create a plain database file from an existing one encrypted with our old/deprecated/unsupported format (<1.18.4413) - then call ChangeSQLEncryptTablePassword() to convert to the new safer format

Variables implemented in the SynSQLite3Static unit

ForceSQLite3LegacyAES: boolean;

Global flag to use initial AES encryption scheme
- IV derivation was hardened in revision 1.18.4607 - set TRUE to this global constant to use the former implementation (theoretically slightly less resistant to brute force attacks) and convert existing databases
27.38. SynSSPI.pas unit

Purpose: Low level access to Windows SSPI/SChannel API for the Win32/Win64 platform - this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Objects implemented in the SynSSPI unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESynSSPI</td>
<td>Exception class raised during SSPI/SChannel process</td>
<td>1715</td>
</tr>
<tr>
<td>SecPkgContext_NamesW</td>
<td>Store the name associated with the context</td>
<td>1715</td>
</tr>
<tr>
<td>TSChannel_Cred</td>
<td>SChannel credential information</td>
<td>1715</td>
</tr>
<tr>
<td>TSecBuffer</td>
<td>Defines a SSPI buffer</td>
<td>1715</td>
</tr>
<tr>
<td>TSecBufferDesc</td>
<td>Describes a SSPI buffer</td>
<td>1715</td>
</tr>
<tr>
<td>TSecContext</td>
<td>SSPI context</td>
<td>1715</td>
</tr>
<tr>
<td>TSecHandle</td>
<td>SSPI context handle</td>
<td>1715</td>
</tr>
<tr>
<td>TSecPkgContext_NegotiationInfo</td>
<td>Store negotiation information about a SSPI package</td>
<td>1715</td>
</tr>
<tr>
<td>TSecPkgContext_Sizes</td>
<td>Store various working buffer sizes of a SSPI command</td>
<td>1715</td>
</tr>
<tr>
<td>TSecPkgContext_StreamSizes</td>
<td>Store various working buffer sizes of a SSPI stream</td>
<td>1715</td>
</tr>
<tr>
<td>TSecPkgCred_SupportedAlgs</td>
<td>Information about SSPI supported algorithm</td>
<td>1715</td>
</tr>
<tr>
<td>TSecPkgInfoW</td>
<td>Store information about a SSPI package</td>
<td>1715</td>
</tr>
<tr>
<td>TSecWinntAuthIdentityW</td>
<td>Information about SSPI Authority Identity</td>
<td>1715</td>
</tr>
<tr>
<td>TSynSSPIAbstract</td>
<td>Abstract parent class for SSPI / SChannel process</td>
<td>1715</td>
</tr>
</tbody>
</table>
TSecHandle = record
  SSPI context handle
end;

TSecContext = record
  SSPI context
end;

TSecBuffer = object(TObject)
  Defines a SSPI buffer
end;

TSecBufferDesc = object(TObject)
  Describes a SSPI buffer
end;

SecPkgContext_Name = record
  Store the name associated with the context
end;

SecPkgInfo = record
  Store information about a SSPI package
end;

SecPkgContext_NegotiationInfo = record
  Store negotiation information about a SSPI package
end;

SecPkgContext_Size = record
  Store various working buffer sizes of a SSPI command
end;

SecPkgContext_StreamSizes = record
  Store various working buffer sizes of a SSPI stream
end;

SecPkgCred_SupportedAlgs = record
  Information about SSPI supported algorithm
end;

SecWinntAuthIdentityW = record
  Information about SSPI Authority Identify
end;

TSChannel_Cred = record
  SChannel credential information
end;

ESynSSPI = class(Exception)
  Exception class raised during SSPI/SChannel process
end;

TSynSSPIAbstract = class(TObject)
  Abstract parent class for SSPI / SChannel process

  constructor Create(aConnectionID: Int64); virtual;
    Initialize the process

  property ConnectionID: Int64 read fContext.ID;
    Read-only access to the associated connection ID, as provided to Create

  property TLS: TSynSSPIModes read fTLS write fTLS;
    The TLS modes supported by this instance
    - only TLS 1.2 is supported by default, for security reasons

Types implemented in the SynSSPI unit

PSChannel_Cred = ^TSChannel_Cred;
Pointer to SChannel credential information

PSecPkgCred_SupportedAlgs = ^TSecPkgCred_SupportedAlgs;

Pointer to SSPI supported algorithm

PSecPkgInfoW = ^TSecPkgInfoW;

Pointer to information about a SSPI package

PSecWinntAuthIdentityW = ^TSecWinntAuthIdentityW;

Pointer to SSPI Authority Identify

TSecContextDynArray = array of TSecContext;

Dynamic array of SSPI contexts
- used to hold information between calls to ServerSSPIAuth

TSynSSPILog = procedure(const Fmt: TSSPIBuffer; const Args: array of const) of object;

Used for low-level logging

TSynSSPIMode = (tls10, tls11, tls12);

The supported TLS modes
- unsafe deprecated modes (e.g. SSL) are not defined at all

TSynSSPIModes = set of TSynSSPIMode;

Set of supported TLS modes

Constants implemented in the SynSSPI unit

SP_PROT_TLS1_2_SERVER = $00000400;

TLS 1.2 should be the preferred safe default

Functions or procedures implemented in the SynSSPI unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeSecContext</td>
<td>Free aSecContext on client or server side</td>
<td>1716</td>
</tr>
<tr>
<td>InvalidateSecContext</td>
<td>Sets aSecHandle fields to empty state for a given connection ID</td>
<td>1716</td>
</tr>
<tr>
<td>SecDecrypt</td>
<td>Decrypts a message</td>
<td>1717</td>
</tr>
<tr>
<td>SecEncrypt</td>
<td>Encrypts a message</td>
<td>1717</td>
</tr>
</tbody>
</table>

procedure FreeSecContext(var aSecContext: TSecContext);
   Free aSecContext on client or server side

procedure InvalidateSecContext(var aSecContext: TSecContext; aConnectionID: Int64);
   Sets aSecHandle fields to empty state for a given connection ID
function SecDecrypt(var aSecContext: TSecContext; const aEncrypted: TSSPIBuffer): TSSPIBuffer;

Decrypts a message
- aSecContext must be set e.g. from previous success call to ServerSSPIAuth or ClientSSPIAuth
- aEncrypted contains data that must be decrypted
- returns decrypted message

function SecEncrypt(var aSecContext: TSecContext; const aPlain: TSSPIBuffer): TSSPIBuffer;

Encrypts a message
- aSecContext must be set e.g. from previous success call to ServerSSPIAuth or ClientSSPIAuth
- aPlain contains data that must be encrypted
- returns encrypted message
27.39. SynSSPIAuth.pas unit

*Purpose:* Low level access to Windows Authentication for the Win32/Win64 platform
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynSSPIAuth unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
<tr>
<td>SynSSPI</td>
<td>Low level access to Windows SSPI/SChannel API for the Win32/Win64 platform - this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1714</td>
</tr>
</tbody>
</table>

**Constants implemented in the SynSSPIAuth unit**

```
SEC_PKG_NAME_NTLM = 'NTLM';

SSPI package names. Client always use Negotiate. Server detect Negotiate or NTLM requests and use appropriate package
```

**Functions or procedures implemented in the SynSSPIAuth unit**

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientForceSPN</td>
<td>Force using aSecKerberosSPN for server identification.</td>
<td>1718</td>
</tr>
<tr>
<td>ClientSSPIAuth</td>
<td>Client-side authentication procedure</td>
<td>1719</td>
</tr>
<tr>
<td>ClientSSPIAuthWithPassword</td>
<td>Client-side authentication procedure with clear text password. This function must be used when application need to use different user credentials (not credentials of logged in user)</td>
<td>1719</td>
</tr>
<tr>
<td>SecPackageName</td>
<td>Returns name of the security package that has been used with the negotiation process</td>
<td>1719</td>
</tr>
<tr>
<td>ServerForceNTLM</td>
<td>Force NTLM authentication instead of Negotiate for browser authentication. Use case: SPNs not configured properly in domain</td>
<td>1719</td>
</tr>
<tr>
<td>ServerSSPIAuth</td>
<td>Server-side authentication procedure</td>
<td>1719</td>
</tr>
<tr>
<td>ServerSSPIAuthUser</td>
<td>Server-side function that returns authenticated user name</td>
<td>1719</td>
</tr>
</tbody>
</table>

```procedure` ClientForceSPN(const aSecKerberosSPN: RawUTF8);`  
*Force using aSecKerberosSPN for server identification.*  
- aSecKerberosSPN is the Service Principal Name, registered in domain, e.g.  
'`mymormotservice/myserver.mydomain.tld@MYDOMAIN.TLD'`
function ClientSSPIAuth(var aSecContext: TSecContext; const aInData: RawByteString; const aSecKerberosSPN: RawUTF8; out aOutData: RawByteString): Boolean;

Client-side authentication procedure
- aSecContext holds information between function calls
- alnData contains data received from server
- aSecKerberosSPN is the optional SPN domain name, e.g. 'mymormotservice/myservice.mydomain.tld'
- aOutData contains data that must be sent to server
- if function returns True, client must send aOutData to server and call function again width data, returned from server

function ClientSSPIAuthWithPassword(var aSecContext: TSecContext; const aInData: RawByteString; const aUserName: RawUTF8; const aPassword: RawUTF8; out aOutData: RawByteString): Boolean;

Client-side authentication procedure with clear text password. This function must be used when application need to use different user credentials (not credentials of logged in user)
- aSecContext holds information between function calls
- alnData contains data received from server
- aUserName is the domain and user name, in form of 'DomainName\UserName'
- aPassword is the user clear text password
- aOutData contains data that must be sent to server
- if function returns True, client must send aOutData to server and call function again width data, returned from server

function SecPackageName(var aSecContext: TSecContext): RawUTF8;

Returns name of the security package that has been used with the negotiation process
- aSecContext must be received from previos success call to ServerSSPIAuth or ClientSSPIAuth

procedure ServerForceNTLM(IsNTLM: boolean);

Force NTLM authentication instead of Negotiate for browser authenticaton. Use case: SPNs not configured properly in domain
- see for details https://synopse.info/forum/viewtopic.php?id=931&p=3

function ServerSSPIAuth(var aSecContext: TSecContext; const aInData: RawByteString; out aOutData: RawByteString): Boolean;

Server-side authentication procedure
- aSecContext holds information between function calls
- alnData contains data received from client
- aOutData contains data that must be sent to client
- if function returns True, server must send aOutData to client and call function again width data, returned from client

procedure ServerSSPIAuthUser(var aSecContext: TSecContext; out aUserName: RawUTF8);

Server-side function that returns authenticated user name
- aSecContext must be received from previous successful call to ServerSSPIAuth
- aUserName contains authenticated user name

Variables implemented in the SynSSPIAuth unit

SECPKGNAMEHTTPAUTHORIZATION: PAnsiChar;

HTTP header pattern received for SSPI authentication 'AUTHORIZATION: NTLM ' or 'AUTHORIZATION: NEGOTIATE '
SECPKGNAMEHTTPWMAUTHENTICATE: RawUTF8;

HTTP header to be set for SSPI authentication 'WWW-Authenticate: NTLM' or 'WWW-Authenticate: Negotiate';
27.40. SynTable.pas unit

*Purpose*: Filter/database/cache/buffer/security/search/multithread/OS features
- as a complement to SynCommons, which tended to increase too much
- licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynTable unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>
# SynTable class hierarchy

## Objects implemented in the SynTable unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EExprParser</td>
<td>Exception type used by TExprParser</td>
<td>1764</td>
</tr>
<tr>
<td>ESynParallelProcess</td>
<td>An exception which would be raised by TSynParallelProcess</td>
<td>1770</td>
</tr>
<tr>
<td>ETableDataException</td>
<td>Exception raised by all TSynTable related code</td>
<td>1784</td>
</tr>
<tr>
<td>ICommandLine</td>
<td>An interface to process the command line switches over a console</td>
<td>1782</td>
</tr>
<tr>
<td>TBlockingProcess</td>
<td>A semaphore used to wait for some process to be finished</td>
<td>1773</td>
</tr>
<tr>
<td>TBlockingProcessPool</td>
<td>Manage a pool of TBlockingProcessPoolItem instances</td>
<td>1774</td>
</tr>
<tr>
<td>TBlockingProcessPoolItem</td>
<td>A semaphore used in the TBlockingProcessPool</td>
<td>1774</td>
</tr>
<tr>
<td>TCommandLine</td>
<td>A class to process the command line switches, with console interactivity</td>
<td>1783</td>
</tr>
<tr>
<td>TDiskPartition</td>
<td>Stores information about a disk partition</td>
<td>1778</td>
</tr>
<tr>
<td>TExprNode</td>
<td>Stores an expression search engine node, as used by TExprParser</td>
<td>1764</td>
</tr>
<tr>
<td>TExprNodeWordAbstract</td>
<td>Abstract class to handle word search, as used by TExprParser</td>
<td>1765</td>
</tr>
<tr>
<td>TExprParserAbstract</td>
<td>Abstract class to parse a text expression into nodes</td>
<td>1765</td>
</tr>
<tr>
<td>TExprParserMatch</td>
<td>Search expression engine using TMatch for the actual word searches</td>
<td>1765</td>
</tr>
<tr>
<td>TFastReader</td>
<td>Safe decoding of a TFileBufferWriter content</td>
<td>1737</td>
</tr>
<tr>
<td>TFileBufferReader</td>
<td>This structure can be used to speed up reading from a file</td>
<td>1745</td>
</tr>
<tr>
<td>TFileBufferWriter</td>
<td>This class can be used to speed up writing to a file</td>
<td>1741</td>
</tr>
<tr>
<td>TIPBan</td>
<td>Optimized thread-safe storage of a list of IP v4 addresses</td>
<td>1764</td>
</tr>
<tr>
<td>TJJSONWriter</td>
<td>Simple writer to a Stream, specialized for the JSON format and SQL export</td>
<td>1736</td>
</tr>
<tr>
<td>TMatch</td>
<td>Low-level structure used by IsMatch() for actual glob search</td>
<td>1727</td>
</tr>
<tr>
<td>TMatches</td>
<td>Stores several TMatch instances, from a set of glob patterns</td>
<td>1728</td>
</tr>
<tr>
<td>TMatchStore</td>
<td>TMatch descendant owning a copy of the Pattern string to avoid GPF issues</td>
<td>1728</td>
</tr>
<tr>
<td>TMemoryInfo</td>
<td>Hold low-level information about current memory usage</td>
<td>1780</td>
</tr>
<tr>
<td>TObjectListSorted</td>
<td>Maintain a thread-safe sorted list of TSynPersistentLock objects</td>
<td>1753</td>
</tr>
<tr>
<td>TParserAbstract</td>
<td>Parent class of TExprParserAbstract</td>
<td>1765</td>
</tr>
<tr>
<td>TPendingTaskList</td>
<td>Handle a list of tasks, stored as RawByteString, with a time stamp</td>
<td>1766</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>TPendingTaskListItem</td>
<td>Internal item definition, used by TPendingTaskList storage</td>
<td>1766</td>
</tr>
<tr>
<td>TProcessInfo</td>
<td>Low-level structure used to compute process memory and CPU usage</td>
<td>1775</td>
</tr>
<tr>
<td>TPropNameList</td>
<td>Simple stack-allocated type for handling a non-void type names list</td>
<td>1757</td>
</tr>
<tr>
<td>TRawByteStringGroup</td>
<td>Store several RawByteString content with optional concatenation</td>
<td>1755</td>
</tr>
<tr>
<td>TRawByteStringGroupVal</td>
<td>Item as stored in a TRawByteStringGroup instance</td>
<td>1755</td>
</tr>
<tr>
<td>TSortCompareTmp</td>
<td>Internal value used by TSynTableFieldProperties.SortCompare() method to avoid stack allocation</td>
<td>1787</td>
</tr>
<tr>
<td>TSQLVar</td>
<td>Memory structure used for database values by reference storage</td>
<td>1736</td>
</tr>
<tr>
<td>TSynAuthentication</td>
<td>Simple authentication class, implementing safe token/challenge security</td>
<td>1763</td>
</tr>
<tr>
<td>TSynAuthenticationAbs</td>
<td>Abstract authentication class, implementing safe token/challenge security and a list of active sessions</td>
<td>1762</td>
</tr>
<tr>
<td>TSynBackgroundThreadAb</td>
<td>Abstract TThread with its own execution content</td>
<td>1767</td>
</tr>
<tr>
<td>TSynBackgroundThreadEv</td>
<td>Allow background thread process of a method callback</td>
<td>1769</td>
</tr>
<tr>
<td>TSynBackgroundThreadMe</td>
<td>Allow background thread process of a variable TThreadMethod callback</td>
<td>1769</td>
</tr>
<tr>
<td>TSynFilter</td>
<td>Will define a transformation to be applied to a Record field content (typicaly a TSQLRecord)</td>
<td>1734</td>
</tr>
<tr>
<td>TSynFilterLowerCase</td>
<td>Convert the value into ASCII Lower Case characters</td>
<td>1734</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TSynFilterLowerCaseU</td>
<td>Convert the value into WinAnsi Lower Case characters</td>
<td>1735</td>
</tr>
<tr>
<td>TSynFilterOrValidate</td>
<td>Will define a filter (transformation) or a validation process to be applied to a database Record content (typically a TSQLRecord)</td>
<td>1729</td>
</tr>
<tr>
<td>TSynFilterTrim</td>
<td>Trim any space character left or right to the value</td>
<td>1735</td>
</tr>
<tr>
<td>TSynFilterTruncate</td>
<td>Truncate a text above a given maximum length</td>
<td>1735</td>
</tr>
<tr>
<td>TSynFilterUpperCase</td>
<td>Convert the value into ASCII Upper Case characters</td>
<td>1734</td>
</tr>
<tr>
<td>TSynFilterUpperCaseU</td>
<td>Convert the value into WinAnsi Upper Case characters</td>
<td>1734</td>
</tr>
<tr>
<td>TSynMonitorDisk</td>
<td>Value object able to gather information about a system drive</td>
<td>1779</td>
</tr>
<tr>
<td>TSynMonitorMemory</td>
<td>Value object able to gather information about the current system memory</td>
<td>1778</td>
</tr>
<tr>
<td>TSynParallelProcess</td>
<td>Allow parallel execution of an index-based process in a thread pool</td>
<td>1770</td>
</tr>
<tr>
<td>TSynParallelProcessThread</td>
<td>Thread executing process for TSynParallelProcess</td>
<td>1770</td>
</tr>
<tr>
<td>TSynPersistentStore</td>
<td>Abstract high-level handling of (SynLZ-)compressed persisted storage</td>
<td>1754</td>
</tr>
<tr>
<td>TSynPersistentStoreJson</td>
<td>Implement binary persistence and JSON serialization (not deserialization)</td>
<td>1755</td>
</tr>
<tr>
<td>TSynPersistentWithPassword</td>
<td>Abstract TSynPersistent class allowing safe storage of a password</td>
<td>1760</td>
</tr>
<tr>
<td>TSynQueue</td>
<td>Thread-safe FIFO (First-In-First-Out) in-order queue of records</td>
<td>1751</td>
</tr>
<tr>
<td>TSynSoundEx</td>
<td>Fast search of a text value, using the Soundex approximation mechanism</td>
<td>1729</td>
</tr>
<tr>
<td>TSynTable</td>
<td>Store the description of a table with records, to implement a Database</td>
<td>1794</td>
</tr>
<tr>
<td>TSynTableData</td>
<td>Used to store a TSynTable record using our SBF compact binary format</td>
<td>1791</td>
</tr>
<tr>
<td>TSynTableFieldProperties</td>
<td>Store the type properties of a given field / database column</td>
<td>1788</td>
</tr>
<tr>
<td>TSynTableStatement</td>
<td>Used to parse a SELECT SQL statement, following the SQLite3 syntax</td>
<td>1786</td>
</tr>
<tr>
<td>TSynTableStatementSelect</td>
<td>One recognized SELECT expression for TSynTableStatement</td>
<td>1785</td>
</tr>
<tr>
<td>TSynTableStatementWhere</td>
<td>One recognized WHERE expression for TSynTableStatement</td>
<td>1785</td>
</tr>
<tr>
<td>TSynTableVariantType</td>
<td>A custom variant type used to have direct access to a record content</td>
<td>1797</td>
</tr>
<tr>
<td>TSynTempWriter</td>
<td>Implements a stack-based writable storage of binary content</td>
<td>1740</td>
</tr>
<tr>
<td>TSynTimeZone</td>
<td>Handle cross-platform time conversions, following Microsoft time zones</td>
<td>1780</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>TSynUniqueIdentifierBits</td>
<td>Map 64-bit integer unique identifier internal memory structure</td>
<td>1758</td>
</tr>
<tr>
<td>TSynUniqueIdentifierGenerator</td>
<td>Thread-safe 64-bit integer unique identifier computation</td>
<td>1759</td>
</tr>
<tr>
<td>TSynUserPassword</td>
<td>Could be used to store a credential pair, as user name and password</td>
<td>1761</td>
</tr>
<tr>
<td>TSynValidate</td>
<td>Will define a validation to be applied to a Record (typically a TSQLRecord) field content</td>
<td>1730</td>
</tr>
<tr>
<td>TSynValidateEmail</td>
<td>IP address validation to be applied to a Record field content (typically a TSQLRecord)</td>
<td>1730</td>
</tr>
<tr>
<td>TSynValidateIPAddress</td>
<td>IP v4 address validation to be applied to a Record field content (typically a TSQLRecord)</td>
<td>1730</td>
</tr>
<tr>
<td>TSynValidateNonVoidText</td>
<td>Text validation to ensure that to any text field would not be ''</td>
<td>1732</td>
</tr>
<tr>
<td>TSynValidatePassword</td>
<td>Strong password validation for a Record field content (typically a TSQLRecord)</td>
<td>1734</td>
</tr>
<tr>
<td>TSynValidatePattern</td>
<td>Glob case-sensitive pattern validation of a Record field content</td>
<td>1731</td>
</tr>
<tr>
<td>TSynValidatePatternInsensitive</td>
<td>Glob case-insensitive pattern validation of a text field content (typically a TSQLRecord)</td>
<td>1731</td>
</tr>
<tr>
<td>TSynValidateTable</td>
<td>Will define a validation to be applied to a TSynTableFieldProperties field</td>
<td>1793</td>
</tr>
<tr>
<td>TSynValidateTableUniqueField</td>
<td>Will define a validation for a TSynTableFieldProperties Unique field</td>
<td>1793</td>
</tr>
<tr>
<td>TSynValidateText</td>
<td>To have existing RTTI for published properties text validation to be applied to any Record field content</td>
<td>1732</td>
</tr>
<tr>
<td>TSystemUse</td>
<td>Monitor CPU and RAM usage of one or several processes</td>
<td>1775</td>
</tr>
<tr>
<td>TSystemUseData</td>
<td>Store CPU and RAM usage for a given process</td>
<td>1775</td>
</tr>
<tr>
<td>TSystemUseProcess</td>
<td>Internal storage of CPU and RAM usage for one process</td>
<td>1775</td>
</tr>
<tr>
<td>TTimeZoneData</td>
<td>Used to store Time Zone information for a single area in TSynTimeZone</td>
<td>1780</td>
</tr>
<tr>
<td>TTimeZoneInfo</td>
<td>Used to store Time Zone bias in TSynTimeZone</td>
<td>1780</td>
</tr>
<tr>
<td>TUpdateFieldEvent</td>
<td>An opaque structure used for TSynTable.UpdateFieldEvent method</td>
<td>1792</td>
</tr>
</tbody>
</table>
TMatch = object(TObject)

Low-level structure used by IsMatch() for actual glob search
- you can use this object to prepare a given pattern, e.g. in a loop
- implemented as a fast brute-force state-machine without any heap allocation
- some common patterns ('exactmatch', 'startwith*', '*endwith', '*contained*') are handled with
dedicated code, optionally with case-insensitive search
- consider using TMatchs (or SetMatchs/TMatchDynArray) if you expect to search for several
patterns, or even TExprParserMatch for expression search

Search: TMatchSearchFunction;
Published for proper inlining

function Equals(const aAnother: TMatch): boolean;
Returns TRUE if this search pattern matches another

function Match(const aText: RawUTF8): boolean; overload;
Returns TRUE if the supplied content matches the prepared glob pattern
- this method is not thread-safe

function Match(aText: PUTF8Char; aTextLen: PInt): boolean; overload;
Returns TRUE if the supplied content matches the prepared glob pattern
- this method is not thread-safe

function MatchString(const aText: string): boolean;
Returns TRUE if the supplied VCL/LCL content matches the prepared glob pattern
- this method IS thread-safe, will use stack to UTF-8 temporary conversion if possible, and won't
lock

function MatchThreadSafe(const aText: RawUTF8): boolean;
Returns TRUE if the supplied content matches the prepared glob pattern
- this method IS thread-safe, and won't lock

function PatternLength: integer;
Access to the pattern length as stored in PMax + 1

function PatternText: PUTF8Char;
Access to the pattern text as stored in Pattern

procedure Prepare(aPattern: PUTF8Char; aPatternLen: integer; aCaseInsensitive, aReuse: boolean); overload;
Initialize the internal fields for a given glob search pattern
- note that the aPattern buffer should remain in memory, since it will be pointed to by the
Pattern private field of this object

procedure Prepare(const aPattern: RawUTF8; aCaseInsensitive, aReuse: boolean); overload;
Initialize the internal fields for a given glob search pattern
- note that the aPattern instance should remain in memory, since it will be pointed to by the
Pattern private field of this object
procedure PrepareContains(var aPattern: RawUTF8; aCaseInsensitive: boolean);
overload;

Initialize low-level internal fields for "aPattern" search
- this method is faster than a regular Prepare("" + aPattern + ")
- warning: the supplied aPattern variable may be modified in-place to be filled with some lookup
buffer, for length(aPattern) in [2..31] range

procedure PrepareRaw(aPattern: PUTF8Char; aPatternLen: integer; aSearch: TMatchSearchFunction);

Initialize low-level internal fields for a custom search algorithm

TMatchStore = record
TMatch descendant owning a copy of the Pattern string to avoid GPF issues

Pattern: TMatch;

Access to the research criteria
- defined as a nested record (and not an object) to circumvent Delphi bug

PatternInstance: RawUTF8;

Pattern.Pattern PUTF8Char will point to this instance

TMatches = class(TSynPersistent)
Stores several TMatch instances, from a set of glob patterns

constructor Create(const aPatterns: TRawUTF8DynArray; CaseInsensitive: Boolean); reintroduce; overload;

Add once some glob patterns to the internal TMach list
- aPatterns[] follows the IsMatch() syntax

function Match(aText: PUTF8Char; aLen: integer): integer; overload;
Search patterns in the supplied UTF-8 text buffer

function Match(const aText: RawUTF8): integer; overload;
Search patterns in the supplied UTF-8 text
- returns -1 if no filter has been subscribed
- returns -2 if there is no match on any previous pattern subscription
- returns fMatch[] index, i.e. >= 0 number on first matching pattern
- this method is thread-safe

function MatchString(const aText: string): integer;
Search patterns in the supplied VCL/LCL text
- could be used on a TFileName for instance
- will avoid any memory allocation if aText is small enough

procedure Subscribe(const aPatterns: TRawUTF8DynArray; CaseInsensitive: Boolean); overload; virtual;

Add once some glob patterns to the internal TMach list
- aPatterns[] follows the IsMatch() syntax
procedure Subscribe(const aPatternsCSV: RawUTF8; CaseInsensitive: Boolean); overload;

   Add once some glob patterns to the internal TMach list
   - each CSV item in aPatterns follows the IsMatch() syntax

TSynSoundEx = object(TObject)
   Fast search of a text value, using the Soundex approximation mechanism
   - Soundex is a phonetic algorithm for indexing names by sound, as pronounced in a given
     language. The goal is for homophones to be encoded to the same representation so that they can
     be matched despite minor differences in spelling
   - this implementation is very fast and can be used e.g. to parse and search in a huge text buffer
   - this version also handles french and spanish pronunciations on request, which differs from
default Soundex, i.e. English

function Ansi(A: PAnsiChar): boolean;
   Return true if prepared value is contained in a ANSI text buffer by using the SoundEx comparison
   algorithm
   - search prepared value at every word beginning in A^

function Prepare(UpperValue: PAnsiChar; Lang: PSoundExValues): boolean; overload;
   Prepare for a custom Soundex search
   - you can specify any language pronunciation from raw TSoundExValues array

function Prepare(UpperValue: PAnsiChar; Lang: TSynSoundExPronunciation=sndxEnglish): boolean; overload;
   Prepare for a Soundex search
   - you can specify another language pronunciation than default english

function UTF8(U: PUTF8Char): boolean;
   Return true if prepared value is contained in a text buffer (UTF-8 encoded), by using the SoundEx
   comparison algorithm
   - search prepared value at every word beginning in U^

TSynFilterOrValidate = class(TObject)
   Will define a filter (transformation) or a validation process to be applied to a database Record
   content (typically a TSQLRecord)
   - the optional associated parameters are to be supplied JSON-encoded

constructor Create(const aParameters: RawUTF8=''); overload; virtual;
   Initialize the filter (transformation) or validation instance
   - most of the time, optional parameters may be specified as JSON, possibly with the extended
     MongoDB syntax

constructor CreateUTF8(const Format: RawUTF8; const Args, Params: array of const); overload;
   Initialize the filter or validation instance
   - this overloaded constructor will allow to easily set the parameters
function AddOnce(var aObjArray: TSynFilterOrValidateObjArray; aFreeIfAlreadyThere: boolean=true): TSynFilterOrValidate;

Add the filter or validation process to a list, checking if not present
- if an instance with the same class type and parameters is already registered, will call aInstance.Free and return the existing instance
- if there is no similar instance, will add it to the list and return it

property Parameters: RawUTF8 read fParameters write SetParameters;
The optional associated parameters, supplied as JSON-encoded

TSynValidate = class(TSynFilterOrValidate)
Will define a validation to be applied to a Record (typically a TSQLRecord) field content
- a typical usage is to validate an email or IP address e.g.
- the optional associated parameters are to be supplied JSON-encoded

function Process(FieldIndex: integer; const Value: RawUTF8; var ErrorMsg: string): boolean;
virtual; abstract;
Perform the validation action to the specified value
- the value is expected by be UTF-8 text, as generated by TPropInfo.GetValue e.g.
- if the validation failed, must return FALSE and put some message in ErrorMsg (translated into the current language: you could e.g. use a resourcestring and a SysUtils.Format() call for automatic translation via the mORMoti18n unit - you can leave ErrorMsg='' to trigger a generic error message from class name ('"Validate email" rule failed' for TSynValidateEmail class e.g.)
- if the validation passed, will return TRUE

TSynValidateIPAddress = class(TSynValidate)
IP v4 address validation to be applied to a Record field content (typically a TSQLRecord)
- this versions expect no parameter

function Process(aFieldIndex: integer; const Value: RawUTF8; var ErrorMsg: string): boolean;
override;
Perform the IP Address validation action to the specified value

TSynValidateEmail = class(TSynValidate)
IP address validation to be applied to a Record field content (typically a TSQLRecord)
- optional JSON encoded parameters are "AllowedTLD" or "ForbiddenTLD", expecting a CSV list of Top-Level-Domain (TLD) names, e.g.
'{"AllowedTLD":"com,org,net","ForbiddenTLD":"fr"}'
'{"AnyTLD: true,ForbiddenDomains:"mailinator.com,yopmail.com"}"
- this will process a validation according to RFC 822 (calling the IsValidEmail() function) then will check for the TLD to be in one of the Top-Level domains ('.com' and such) or a two-char country, and then will check the TLD according to AllowedTLD and ForbiddenTLD

function Process(aFieldIndex: integer; const Value: RawUTF8; var ErrorMsg: string): boolean;
override;
Perform the Email Address validation action to the specified value
- call IsValidEmail() function and check for the supplied TLD
property AllowedTLD: RawUTF8 read fAllowedTLD write fAllowedTLD;
  A CSV list of allowed TLD
  - if accessed directly, should be set as lower case values
  - e.g. 'com,org,net'

property AnyTLD: boolean read fAnyTLD write fAnyTLD;
  Allow any TLD to be allowed, even if not a generic TLD (.com,.net ...)
  - this may be mandatory since already over 1,300 new gTLD names or "strings" could become
    available in the next few years: there is a growing list of new gTLDs available at
    @http://newgtlds.icann.org/en/program-status/delegated-strings
  - the only restriction is that it should be ascii characters

property ForbiddenDomains: RawUTF8 read fForbiddenDomains write fForbiddenDomains;
  A CSV list of forbidden domain names
  - if accessed directly, should be set as lower case values
  - not only the TLD, but whole domains like 'cracks.ru,hotmail.com' or such

property ForbiddenTLD: RawUTF8 read fForbiddenTLD write fForbiddenTLD;
  A CSV list of forbidden TLD
  - if accessed directly, should be set as lower case values
  - e.g. 'fr'

TSynValidatePattern = class(TSynValidate)
  Glob case-sensitive pattern validation of a Record field content
  - parameter is NOT JSON encoded, but is some basic TMatch glob pattern
  - ? = Matches any single character
  - * = Matches any contiguous characters
  - [abc] = Matches a or b or c at that position
  - [^abc] = Matches anything but a or b or c at that position
  - [abcx-z] = Matches a or b or c or x or y or z, as does [a-cx-z]
  - 'ma?ch.*' = would match match.exe, mavch.dat, march.on, etc..
  - 'this [e-n]s a [!zy]est' would match 'this is a test', but would not match 'this as a test' nor 'this is a zest'
  - pattern check IS case sensitive (TSynValidatePatternI is not)
  - this class is not as complete as PCRE regex for example, but code overhead is very small, and
    speed good enough in practice

function Process(aFieldIndex: integer; const Value: RawUTF8; var ErrorMsg: string): boolean; override;
  Perform the pattern validation to the specified value
  - pattern can be e.g. '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]'
  - this method will implement both TSynValidatePattern and TSynValidatePatternI, checking the
    current class

TSynValidatePatternI = class(TSynValidatePattern)
  Glob case-insensitive pattern validation of a text field content (typicaly a TSQLRecord)
  - parameter is NOT JSON encoded, but is some basic TMatch glob pattern
  - same as TSynValidatePattern, but is NOT case sensitive
TSynValidateNonVoidText = class(TSynValidate)

Text validation to ensure that to any text field would not be "

function Process(aFieldIndex: integer; const Value: RawUTF8; var ErrorMsg: string): boolean; override;

Perform the non void text validation action to the specified value

TSynValidateText = class(TSynValidate)

To have existing RTTI for published properties text validation to be applied to any Record field content
- default MinLength value is 1, MaxLength is maxInt: so a blank TSynValidateText.Create("") is the same as TSynValidateNonVoidText
- MinAlphaCount, MinDigitCount, MinPunctCount, MinLowerCount and MinUpperCount allow you to specify the minimal count of respectively alphabetical [a-zA-Z], digit [0-9], punctuation [_,;:!?@#%=]+-, lower case or upper case characters
- expects optional JSON parameters of the allowed text length range as
  '{"MinLength":5,"MaxLength":10,"MinAlphaCount":1,"MinDigitCount":1,"MinPunctCount":1,"MinLowerCount":1,"MinUpperCase":1}"

function Process(aFieldIndex: integer; const Value: RawUTF8; var ErrorMsg: string): boolean; override;

Perform the text length validation action to the specified value

property MaxAlphaCount: cardinal read fProps[10] write fProps[10];

Maximal alphabetical character [a-zA-Z] count
- default is maxInt, i.e. no Maximum set


Maximal digit character [0-9] count
- default is maxInt, i.e. no Maximum set

property MaxLeftTrimCount: cardinal read fProps[8] write fProps[8];

Maximal space count allowed on the Left side
- default is maxInt, i.e. any Left space allowed

property MaxLength: cardinal read fProps[1] write fProps[1];

Maximal length value allowed for the text content
- the length is calculated with UTF-16 Unicode codepoints, unless UTF8Length has been set to TRUE so that the UTF-8 byte count is checked
- default is maxInt, i.e. no maximum length is set

property MaxLowerCount: cardinal read fProps[13] write fProps[13];

Maximal alphabetical lower case character [a-z] count
- default is maxInt, i.e. no Maximum set

property MaxPunctCount: cardinal read fProps[12] write fProps[12];

Maximal punctuation sign [_-;:!?@#%=]+- count
- default is maxInt, i.e. no Maximum set
property MaxRightTrimCount: cardinal read fProps[9] write fProps[9];
Maximal space count allowed on the Right side
- default is maxInt, i.e. any Right space allowed

property MaxSpaceCount: cardinal read fProps[15] write fProps[15];
Maximal space count inside the value text
- default is maxInt, i.e. any space number allowed

property MaxUpperCount: cardinal read fProps[14] write fProps[14];
Maximal alphabetical upper case character [A-Z] count
- default is maxInt, i.e. no Maximum set

property MinAlphaCount: cardinal read fProps[2] write fProps[2];
Minimal alphabetical character [a-zA-Z] count
- default is 0, i.e. no minimum set

property MinLength: cardinal read fProps[0] write fProps[0];
Minimal length value allowed for the text content
- the length is calculated with UTF-16 Unicode codepoints, unless UTF8Length has been set to TRUE so that the UTF-8 byte count is checked
- default is 1, i.e. a void text will not pass the validation

property MinLowerCount: cardinal read fProps[5] write fProps[5];
Minimal alphabetical lower case character [a-z] count
- default is 0, i.e. no minimum set

property MinPunctCount: cardinal read fProps[4] write fProps[4];
Minimal punctuation sign [,_;/?:.?!@$%^&*(){}+-] count
- default is 0, i.e. no minimum set

property MinSpaceCount: cardinal read fProps[7] write fProps[7];
Minimal space count inside the value text
- default is 0, i.e. any space number allowed

property MinUpperCount: cardinal read fProps[6] write fProps[6];
Minimal alphabetical upper case character [A-Z] count
- default is 0, i.e. no minimum set

property UTF8Length: boolean read fUTF8Length write fUTF8Length;
Defines if lengths parameters expects UTF-8 or UTF-16 codepoints number
- with default FALSE, the length is calculated with UTF-16 Unicode codepoints - MaxLength may not match the UCS4 glyphs number, in case of UTF-16 surrogates
- you can set this property to TRUE so that the UTF-8 byte count would be used for truncation againtosh the MaxLength parameter
TSynValidatePassword = class(TSynValidateText)

- Strong password validation for a Record field content (typically a TSQLRecord)
  - the following parameters are set by default to
    '{"MinLength":5,"MaxLength":20,"MinAlphaCount":1,"MinDigitCount":1,
    "MinPunctCount":1,"MinLowerCount":1,"MinUpperCount":1,"MaxSpaceCount":0}'
  - you can specify some JSON encoded parameters to change this default values, which will validate
    the text field only if it contains from 5 to 10 characters, with at least one digit, one upper case
    letter, one lower case letter, and one punctuation sign, with no space allowed inside

TSynFilter = class(TSynFilterOrValidate)

Will define a transformation to be applied to a Record field content (typically a TSQLRecord)
- here "filter" means that content would be transformed according to a set of defined rules
- a typical usage is to convert to lower or upper case, or trim any time or date value in a
  TDateTime field
- the optional associated parameters are to be supplied JSON-encoded

procedure Process(aFieldIndex: integer; var Value: RawUTF8); virtual; abstract;
  Perform the transformation to the specified value
  - the value is converted into UTF-8 text, as expected by TPropInfo.GetValue / TPropInfo.SetValue
  e.g.

TSynFilterUpperCase = class(TSynFilter)

Convert the value into ASCII Upper Case characters
- UpperCase conversion is made for ASCII-7 only, i.e. 'a'..'z' characters
- this version expects no parameter

procedure Process(aFieldIndex: integer; var Value: RawUTF8); override;
  Perform the case conversion to the specified value

TSynFilterUpperCaseU = class(TSynFilter)

Convert the value into WinAnsi Upper Case characters
- UpperCase conversion is made for all latin characters in the WinAnsi code page only, e.g. 'e'
  acute will be converted to 'E'
- this version expects no parameter

procedure Process(aFieldIndex: integer; var Value: RawUTF8); override;
  Perform the case conversion to the specified value

TSynFilterLowerCase = class(TSynFilter)

Convert the value into ASCII Lower Case characters
- LowerCase conversion is made for ASCII-7 only, i.e. 'A'..'Z' characters
- this version expects no parameter

procedure Process(aFieldIndex: integer; var Value: RawUTF8); override;
  Perform the case conversion to the specified value
**TSynFilterLowerCaseU** = class(TSynFilter)

*Convert the value into WinAnsi Lower Case characters*
- LowerCase conversion is made for all latin characters in the WinAnsi code page only, e.g. 'E' acute will be converted to 'e'
- this version expects no parameter

*procedure* Process(aFieldIndex: integer; var Value: RawUTF8); override;

*Perform the case conversion to the specified value*

**TSynFilterTrim = class(TSynFilter)**

*Trim any space character left or right to the value*
- this versions expect no parameter

*procedure* Process(aFieldIndex: integer; var Value: RawUTF8); override;

*Perform the space triming conversion to the specified value*

**TSynFilterTruncate = class(TSynFilter)**

*Truncate a text above a given maximum length*
- expects optional JSON parameters of the allowed text length range as

`'{MaxLength":10}`

*procedure* Process(aFieldIndex: integer; var Value: RawUTF8); override;

*Perform the length truncation of the specified value*

**property** MaxLength: cardinal read fMaxLength write fMaxLength;

*Maximum length value allowed for the text content*
- the length is calculated with UTF-16 Unicode codepoints, unless UTF8Length has been set to TRUE so that the UTF-8 byte count is checked
- default is 0, i.e. no maximum length is forced

**property** UTF8Length: boolean read fUTF8Length write fUTF8Length;

*Defines if MaxLength is stored as UTF-8 or UTF-16 codepoints number*
- with default FALSE, the length is calculated with UTF-16 Unicode codepoints - MaxLength may not match the UCS4 glyphs number, in case of UTF-16 surrogates
- you can set this property to TRUE so that the UTF-8 byte count would be used for truncation against the MaxLength parameter
**TSQLVar** = record

* Memory structure used for database values by reference storage
* - used mainly by SynDB, mORMot, mORMotDB and mORMotSQLite3 units
* - defines only TSQLDBFieldType data types (similar to those handled by SQLite3, with the addition of ftCurrency and ftDate)
* - cleaner/lighter dedicated type than TValue or variant/TVarData, strong enough to be marshalled as JSON content
* - variable-length data (e.g. UTF-8 text or binary BLOB) are never stored within this record, but VText/VBlob will point to an external (temporary) memory buffer
* - date/time is stored as ISO-8601 text (with milliseconds if svoDateWithMS option is set and the database supports it), and currency as double or BCD in most databases

**VType:** TSQLDBFieldType

* The type of the value stored

---

**TJSONWriter** = class(TTextWriterWithEcho)

* Simple writer to a Stream, specialized for the JSON format and SQL export
* - i.e. define some property/method helpers to export SQL resultset as JSON
* - see mORMot.pas for proper class serialization via TJSONSerializer.writeObject

* ColNames: TRawUTF8DynArray;

* Used internally to store column names and count for AddColumns

**constructor** Create(aStream: TStream; Expand, withID: boolean; const Fields: TSQLFieldIndexDynArray=nil; aBufSize: integer=8192; aStackBuffer: PTextWriterStackBuffer=nil); overload;

* The data will be written to the specified Stream
* - if no Stream is supplied, a temporary memory stream will be created (it's faster to supply one, e.g. any TSQLRest.TempMemoryStream)

**constructor** Create(aStream: TStream; Expand, withID: boolean; const Fields: TSQLFieldBits; aBufSize: integer=8192); overload;

* The data will be written to the specified Stream
* - if no Stream is supplied, a temporary memory stream will be created (it's faster to supply one, e.g. any TSQLRest.TempMemoryStream)

**procedure** AddColumns(aKnownRowsCount: integer=0);

* Write or init field names for appropriate JSON Expand later use
* - ColNames[] must have been initialized before calling this procedure
* - if aKnownRowsCount is not null, a "rowCount":... item will be added to the generated JSON stream (for faster unserialization of huge content)

**procedure** CancelAllVoid;

* Rewind the Stream position and write void JSON object
procedure ChangeExpandedFields(aWithID: boolean; const aFields: TSQLFieldIndexDynArray); overload;

   Allow to change on the fly an expanded format column layout
   - by definition, a non expanded format will raise a ESynException
   - caller should then set ColNames[] and run AddColumns()

procedure EndJSONObject(aKnownRowsCount, aRowsCount: integer; aFlushFinal: boolean=True);

   End the serialized JSON object
   - cancel last ','
   - close the JSON object } or ]'
   - write non expanded postlog ("rowcount":...), if needed
   - flush the internal buffer content if aFlushFinal=true

procedure TrimFirstRow;

   The first data row is erased from the content
   - only works if the associated storage stream is TMemoryStream
   - expect not Expanded format

property Expand: boolean read fExpand write fExpand;

   Is set to TRUE in case of Expanded format

property Fields: TSQLFieldIndexDynArray read fFields;

   Read-Only access to the field bits set for each column to be stored

property StartDataPosition: integer read fStartDataPosition;

   If not Expanded format, contains the Stream position of the first useful Row of data; i.e. ',val11'
   position in:
   { "fieldCount":1,"values":["col1","col2",val11,"val12",val21,...] }

property WithID: boolean read fWithID;

   Is set to TRUE if the ID field must be appended to the resulting JSON
   - this field is used only by TSQLRecord.GetJSONValues
   - this field is ignored by TSQLTable.GetJSONValues

TFastReader = object(TObject)

   Safe decoding of a TFileBufferWriter content
   - similar to TFileBufferReader, but faster and only for in-memory buffer
   - is also safer, since will check for reaching end of buffer
   - raise a EFastReader exception on decoding error (e.g. if a buffer overflow may occur) or call
     OnErrorOverflow/OnErrorData event handlers

   Last: PAnsiChar;

   The last position in the buffer

   OnErrorData: procedure(const fmt: RawUTF8; const args: array of const) of object;

      Use this event to customize the ErrorData process

   OnErrorOverflow: procedure of object;

      Use this event to customize the ErrorOverflow process
P: PAnsiChar;
   The current position in the memory

Tag: PtrInt;
   Some opaque value, which may be a version number to define the binary layout

function CopySafe(out Dest; DataLen: PtrInt): boolean;
   Copy data from the current position, and move ahead the specified bytes
   - this version won't call ErrorOverflow, but return false on error
   - returns true on read success

function EOF: boolean;
   Returns TRUE if the current position is the end of the input stream

function Next(DataLen: PtrInt): pointer;
   Returns the current position, and move ahead the specified bytes

function Next4: cardinal;
   Read the next 4 bytes from the buffer as a 32-bit unsigned value

function Next8: Qword;
   Read the next 8 bytes from the buffer as a 64-bit unsigned value

function NextByte: byte;
   Read the next byte from the buffer

function NextByteEquals(Value: byte): boolean;
   Consumes the next byte from the buffer, if matches a given value

function NextByteSafe(dest: pointer): boolean;
   Read the next byte from the buffer, checking

function NextSafe(out Data: Pointer; DataLen: PtrInt): boolean;
   Returns the current position, and move ahead the specified bytes

function PeekVarInt32(out Value: PtrInt): boolean;
   Try to read the next 32-bit signed value from the buffer
   - don't change the current position

function PeekVarUInt32(out Value: PtrUInt): boolean;
   Try to read the next 32-bit unsigned value from the buffer
   - don't change the current position

function ReadCompressed(Load: TAlgoCompressLoad=aclNormal; BufferOffset: integer=0): RawByteString;
   Retrieve some TAlgoCompress buffer, appended via Write()
   - BufferOffset could be set to reserve some bytes before the uncompressed buffer

function ReadVarUInt32Array(var Values: TIntegerDynArray):_PTRInt;
   Retrieved cardinal values encoded with TFileBufferWriter.WriteVarUInt32Array
   - only supports wkUInt32, wkVarInt32, wkVarUInt32 kind of encoding

function RemainingLength: PtrUInt;
   Returns remaining length (difference between Last and P)
function VarBlob: TValueResult; overload;
    Read the next pointer and length value from the buffer

function VarInt32: integer;
    Read the next 32-bit signed value from the buffer

function VarInt64: Int64;
    Read the next 64-bit signed value from the buffer

function VarShortString: shortstring;
    Read the next ShortString value from the buffer

function VarString: RawByteString;
    Read the next RawByteString value from the buffer

function VarUInt32: cardinal;
    Read the next 32-bit unsigned value from the buffer

function VarUInt32Safe(out Value: cardinal): boolean;
    Read the next 32-bit unsigned value from the buffer
    - this version won't call ErrorOverflow, but return false on error
    - returns true on read success

function VarUInt64: QWord;
    Read the next 64-bit unsigned value from the buffer

function VarUTF8: RawUTF8; overload;
    Read the next RawUTF8 value from the buffer

function VarUTF8Safe(out Value: RawUTF8): boolean;
    Read the next RawUTF8 value from the buffer
    - this version won't call ErrorOverflow, but return false on error
    - returns true on read success

procedure Copy(out Dest; DataLen: PtrInt);
    Copy data from the current position, and move ahead the specified bytes

procedure ErrorData(const fmt: RawUTF8; const args: array of const);
    Raise a EFastReader with an "incorrect data" error message

procedure ErrorOverflow;
    Raise a EFastReader with an "overflow" error message

procedure Init(const Buffer: RawByteString); overload;
    Initialize the reader from a RawByteString content

procedure Init(Buffer: pointer; Len: integer); overload;
    Initialize the reader from a memory block

procedure NextDocVariantData(out Value: variant; CustomVariantOptions: PDocVariantOptions);
    Read the JSON-serialized TDocVariant from the buffer
    - matches TFileBufferWriter.WriteDocVariantData format
procedure NextVariant(var Value: variant; CustomVariantOptions: PDocVariantOptions);
    // Read the next variant from the buffer
    // - is a wrapper around VariantLoad()

procedure Read(var DA: TDynArray; NoCheckHash: boolean=false);
    // Apply TDynArray.LoadFrom on the buffer
    // - will unserialize a previously appended dynamic array, e.g. as
    //   aWriter.WriteDynArray(DA);

procedure VarBlob(out result: TValueResult); overload;
    // Read the next pointer and length value from the buffer

procedure VarNextInt(count: integer); overload;
    // Fast ignore the next count VarUInt32/VarInt32/VarUInt64/VarInt64 values
    // - don’t raise any exception, so caller could check explicitly for any EOF

procedure VarNextInt; overload;
    // Fast ignore the next VarUInt32/VarInt32/VarUInt64/VarInt64 value
    // - don’t raise any exception, so caller could check explicitly for any EOF

procedure VarUTF8(out result: RawUTF8); overload;
    // Read the next RawUTF8 value from the buffer

TSynTempWriter = object(TObject)
    // Implements a stack-based writable storage of binary content
    // - memory allocation is performed via a TSynTempBuffer
    pos: PAnsiChar;
        // The current writable position in tmp.buf

function AsBinary: RawByteString;
    // Returns the buffer as a RawByteString instance

function Position: PtrInt;
    // Returns the current offset position in the internal buffer

function wrfillchar(count: integer; value: byte): PAnsiChar;
    // Append some fixed-value bytes as binary to the internal buffer
    // - returns a pointer to the first byte of the added memory chunk

procedure AsUTF8(var result: RawUTF8);
    // Returns the buffer as a RawUTF8 instance

procedure Done;
    // Finalize the temporary storage

procedure Init(maxsize: integer=0);
    // Initialize a new temporary buffer of a given number of bytes
    // - if maxsize is left to its 0 default value, the default stack-allocated memory size is used, i.e. 4 KB
procedure wr(const val; len: PtrInt);
  Append some binary to the internal buffer
  - will raise an ESynException in case of potential overflow

procedure wrb(b: byte);
  Append some 8-bit value as binary to the internal buffer

procedure wrint(int: integer);
  Append some 32-bit value as binary to the internal buffer

procedure wrptr(ptr: pointer);
  Append some 32-bit/64-bit pointer value as binary to the internal buffer

procedure wrptrint(int: PtrInt);
  Append some 32-bit/64-bit integer as binary to the internal buffer

procedure wrs(const str: RawByteString);
  Append some string as binary to the internal buffer

procedure wrss(const str: shortstring);
  Append some shortstring as binary to the internal buffer

procedure wrw(w: word);
  Append some 16-bit value as binary to the internal buffer

TFileBufferWriter = class(TObject)
  This class can be used to speed up writing to a file
  - big speed up if data is written in small blocks
  - also handle optimized storage of any dynamic array of Integer/Int64/RawUTF8
  - use TFileBufferReader or TFastReader for decoding of the stored binary

constructor Create(aClass: TStreamClass; BufLen: integer=4096); overload;
  Initialize the buffer, using an internal TStream instance
  - parameter could be e.g. THeapMemoryStream or TRawByteStringStream
  - use Flush then TMemoryStream(Stream) to retrieve its content, or
    TRawByteStringStream(Stream).DataString

constructor Create(aStream: TStream; aTempBuf: pointer; aTempLen: integer);
  overload;
  Initialize with a specified buffer and TStream class
  - use a specified external buffer (which may be allocated on stack), to avoid a memory allocation

constructor Create(aClass: TStreamClass; aTempBuf: pointer; aTempLen: integer);
  overload;
  Initialize with a specified buffer
  - use a specified external buffer (which may be allocated on stack), to avoid a memory allocation
  - aStream parameter could be e.g. THeapMemoryStream or TRawByteStringStream

constructor Create(aFile: THandle; BufLen: integer=65536); overload;
  Initialize the buffer, and specify a file handle to use for writing
  - use an internal buffer of the specified size
constructor Create(aStream: TStream; BufLen: integer=65536); overload;
    *Initialize the buffer, and specify a TStream to use for writing*
    - use an internal buffer of the specified size

constructor Create(const aFileName: TFileName; BufLen: integer=65536; Append: boolean=false); overload;
    *Initialize the buffer, and specify a file to use for writing*
    - use an internal buffer of the specified size
    - would replace any existing file by default, unless Append is TRUE

destructor Destroy; override;
    *Release internal TStream (after AssignToHandle call)*
    - warning: an explicit call to Flush is needed to write the data pending in internal buffer

function DirectWritePrepare(len: PtrInt; out tmp: RawByteString): PAnsiChar;
    *Allows to write directly to a memory buffer*
    - caller should specify the maximum possible number of bytes to be written
    - then write the data to the returned pointer, and call DirectWriteFlush

function Flush: Int64;
    *Write any pending data in the internal buffer to the file*
    - after a Flush, it's possible to call FileSeek64(aFile,....)
    - returns the number of bytes written between two Flush method calls

function FlushAndCompress(nocompression: boolean=false; algo: TAlgoCompress=nil; BufferOffset: integer=0): RawByteString;
    *Write any pending data, then call algo.Compress() on the buffer*
    - expect the instance to have been created via
      TFileBufferWriter.Create(TRawByteStringStream)
    - if algo is left to its default nil, will use global AlgoSynLZ
    - features direct compression from internal buffer, if stream was not used
    - BufferOffset could be set to reserve some bytes before the compressed buffer

procedure CancelAll; virtual;
    *Rewind the Stream to the position when Create() was called*
    - note that this does not clear the Stream content itself, just move back its writing position to its initial place

procedure DirectWriteFlush(len: PtrInt; const tmp: RawByteString);
    *Finalize a direct write to a memory buffer*
    - by specifying the number of bytes written to the buffer

procedure Write(const Text: RawByteString); overload;
    *Append some UTF-8 encoded text at the current position*
    - will write the string length (as VarUInt32), then the string content, as expected by the FromVarString() function

procedure Write(Data: pointer; DataLen: PtrInt); overload;
    *Append some data at the current position*

procedure Write(const Value: variant); overload;
    *Append some variant value at the current position*
    - matches FromVarVariant() and VariantSave/VariantLoad format
procedure Write1(Data: Byte);
  Append 1 byte of data at the current position

procedure Write2(Data: Word);
  Append 2 bytes of data at the current position

procedure Write4(Data: integer);
  Append 4 bytes of data at the current position

procedure Write4BigEndian(Data: integer);
  Append 4 bytes of data, encoded as BigEndian, at the current position

procedure Write8(const Data8Bytes);
  Append 8 bytes of data at the current position

procedure WriteBinary(const Data: RawByteString);
  Append some content at the current position
  - will write the binary data, without any length prefix

procedure WriteDocVariantData(const Value: variant);
  Append some TDocVariant value at the current position, as JSON string
  - matches TFastReader.NextDocVariantData format

procedure WriteDynArray(const DA: TDynArray);
  Append some dynamic array at the current position
  - will use the binary serialization as for:
    aWriter.WriteBinary(DA.SaveTo);
  but writing directly into the buffer, if possible

procedure WriteN(Data: Byte; Count: integer);
  Append the same byte a given number of occurrences at the current position

procedure WriteRawUTF8Array(Values: PPtrUIntArray; ValuesCount: integer);
  Append a RawUTF8 array of values, from its low-level memory pointer
  - handled the fixed size strings array case in a very efficient way

procedure WriteRawUTF8DynArray(const Values: TRawUTF8DynArray; ValuesCount: integer);
  Append the RawUTF8 dynamic array
  - handled the fixed size strings array case in a very efficient way

procedure WriteRawUTF8List(List: TRawUTF8List; StoreObjectsAsVarUInt32: Boolean=false);
  Append the RawUTF8List content
  - if StoreObjectsAsVarUInt32 is TRUE, all Objects[] properties will be stored as VarUInt32

procedure WriteRecord(const Rec; RecTypeInfo: pointer);
  Append some record at the current position, with binary serialization
  - will use the binary serialization as for:
    aWriter.WriteBinary(RecordSave(Rec,RecTypeInfo));
  but writing directly into the buffer, if possible

procedure WriteShort(const Text: ShortString);
  Append some UTF-8 encoded text at the current position
  - will write the string length (as VarUInt32), then the string content
procedure WriteStream(aStream: TCustomMemoryStream; aStreamSize: Integer=-1);
   Append a TStream content
   - is StreamSize is left as -1, the Stream.Size is used
   - the size of the content is stored in the resulting stream

procedure WriteVarInt32(Value: PtrInt);
   Append an integer value using 32-bit variable-length integer encoding of the by-two complement
   of the given value

procedure WriteVarInt64(Value: Int64);
   Append an integer value using 64-bit variable-length integer encoding of the by-two complement
   of the given value

procedure WriteVarUInt32(Value: PtrUInt);
   Append a cardinal value using 32-bit variable-length integer encoding

procedure WriteVarUInt32Array(const Values: TIntegerDynArray; ValuesCount: integer; DataLayout: TFileBufferWriterKind);
   Append cardinal values (NONE must be negative!) using 32-bit variable-length integer encoding
   or other specialized algorithm, depending on the data layout

procedure WriteVarUInt32Values(Values: PIntegerArray; ValuesCount: integer; DataLayout: TFileBufferWriterKind);
   Append cardinal values (NONE must be negative!) using 32-bit variable-length integer encoding
   or other specialized algorithm, depending on the data layout

procedure WriteVarUInt64(Value: QWord);
   Append an unsigned integer value using 64-bit variable-length encoding

procedure WriteVarUInt64DynArray(const Values: TInt64DynArray; ValuesCount: integer; Offset: Boolean);
   Append UInt64 values using 64-bit variable length integer encoding
   - if Offset is TRUE, then it will store the difference between two values using 64-bit
     variable-length integer encoding (in this case, a fixed-sized record storage is also handled
     separately)
   - could be decoded later on via TFileBufferReader.ReadVarUInt64Array

procedure WriteXor(New,Old: PAnsiChar; Len: PtrInt; crc: PCardinal=nil);
   Append "New[0..Len-1] xor Old[0..Len-1]" bytes
   - as used e.g. by ZeroCompressXor/TSynBloomFilterDiff.SaveTo

property Stream: TStream read fStream;
   The associated writing stream

property Tag: PtrInt read fTag write fTag;
   Simple property used to store some integer content

property TotalWritten: Int64 read fTotalWritten;
   Get the byte count written since last Flush

SynTable.pas unit - Rev. 1.18
Page 1745 of 2552
TFileBufferReader = object(TObject)

This structure can be used to speed up reading from a file
- use internally memory mapped files for a file up to 2 GB (Windows has problems with memory
mapped files bigger than this size limit - at least with 32-bit executables) - but sometimes,
Windows fails to allocate more than 512 MB for a memory map, because it does lack of
contiguous memory space: in this case, we fall back on direct file reading
- maximum handled file size has no limit (but will use slower direct file reading)
- can handle sophisticated storage layout of TFileBufferWriter for dynamic arrays of
Integer/Int64/RawUTF8
- is defined as an object or as a record, due to a bug in Delphi 2009/2010 compiler (at least): this
structure is not initialized if defined as an object on the stack, but will be as a record:

function CurrentMemory(DataLen: PtrUInt=0; PEnd: PAnsiChar=nil): pointer;
Retrieve the current in-memory pointer
- if file was not memory-mapped, returns nil
- if DataLen>0, will increment the current in-memory position

function CurrentPosition: integer;
Retrieve the current in-memory position
- if file was not memory-mapped, returns -1

function FileSize: Int64;
Read-only access to the global file size

function MappedBuffer: PAnsiChar;
Read-only access to the global mapped buffer binary

function OpenFrom(Stream: TStream): boolean; overload;
Initialize the buffer from an already existing Stream
- accept either TFileStream or TCustomMemoryStream kind of stream

function Read(out Text: RawByteString): integer; overload;
Read some buffer text at the current position
- returns the resulting text length, in bytes

function Read(Data: pointer; DataLen: PtrInt): integer; overload;
Read some bytes from the given reading position
- returns the number of bytes which was read
- if Data is nil, it won't read content but will forward reading position

function Read(out Text: RawUTF8): integer; overload;
Read some UTF-8 encoded text at the current position
- returns the resulting text length, in bytes

function ReadByte: PtrUInt;
Read one byte
- if reached end of file, don't raise any error, but returns 0

function ReadCardinal: cardinal;
Read one cardinal, which was written as fixed length
- if reached end of file, don't raise any error, but returns 0


```pascal
function ReadPointer(DataLen: PtrUInt; var aTempData: RawByteString): pointer;
  Retrieve a pointer to the current position, for a given data length
  - if the data is available in the current memory mapped file, it will just return a pointer to it
  - otherwise (i.e. if the data is split between to 1GB memory map buffers), data will be copied
    into the temporary aTempData buffer before retrieval

function ReadRawUTF8: RawUTF8;
  Read some UTF-8 encoded text at the current position
  - returns the resulting text

function ReadRawUTF8List(List: TRawUTF8List): boolean;
  Retrieve the RawUTF8List content encoded with TFileBufferWriter.WriteRawUTF8List
  - if StoreObjectsAsVarUInt32 was TRUE, all Objects[] properties will be retrieved as VarUInt32

function ReadStream(DataLen: PtrInt=-1): TCustomMemoryStream;
  Create a TMemoryStream instance from the current position
  - the content size is either specified by DataLen>=0, either available at the current position, as
    saved by TFileBufferWriter.WriteStream method
  - if this content fit in the current 1GB memory map buffer, a TSynMemoryStream instance is
    returned, with no data copy (faster)
  - if this content is not already mapped in memory, a separate memory map will be created (the
    returned instance is a TSynMemoryStreamMapped)

function ReadVarInt32: PtrInt;
  Read one integer value encoded using our 32-bit variable-length integer, and the by-two
  complement

function ReadVarInt64: Int64;
  Read one Int64 value encoded using our 64-bit variable-length integer

function ReadVarRawUTF8DynArray(var Values: TRawUTF8DynArray): PtrInt;
  Retrieved RawUTF8 values encoded with TFileBufferWriter.WriteRawUTF8DynArray
  - returns the number of items read into Values[] (may differ from length(Values))

function ReadVarUInt32: PtrUInt;
  Read one cardinal value encoded using our 32-bit variable-length integer

function ReadVarUInt32Array(var Values: TIntegerDynArray): PtrInt;
  Retrieved cardinal values encoded with TFileBufferWriter.WriteVarUInt32Array
  - returns the number of items read into Values[] (may differ from length(Values)), which will be
    resized, so could be void before calling
  - if the returned integer is negative, it is -Count, and testifies from wkFakeMarker and the
    content should be retrieved by the caller

function ReadVarUInt64: QWord;
  Read one UInt64 value encoded using our 64-bit variable-length integer

function ReadVarUInt64Array(var Values: TInt64DynArray): PtrInt;
  Retrieved Int64 values encoded with TFileBufferWriter.WriteVarUInt64DynArray
  - returns the number of items read into Values[] (may differ from length(Values))

function Seek(Offset: PtrInt): boolean; overload;
  Change the current reading position, from the beginning of the file
  - returns TRUE if success, or FALSE if Offset is out of range
```
function Seek(Offset: Int64): boolean; overload;
  Change the current reading position, from the beginning of the file
  - returns TRUE if success, or FALSE if Offset is out of range

procedure Close;
  Close all internal mapped files
  - call Open() again to use the Read() methods

procedure ErrorInvalidContent;
  Raise an exception in case of invalid content

procedure Open(aFile: THandle; aFileNotMapped: boolean=false);
  Initialize the buffer, and specify a file to use for reading
  - will try to map the whole file content in memory
  - if memory mapping failed, or aFileNotMapped is true, methods will use default slower file API

procedure OpenFrom(aBuffer: pointer; aBufferSize: PtrUInt); overload;
  Initialize the buffer from an already existing memory block
  - may be e.g. a resource or a TMemoryStream

procedure OpenFrom(const aBuffer: RawByteString); overload;
  Initialize the buffer from an already existing memory block

TSynBloomFilter = class(TSynPersistentLock)
  Implements a thread-safe Bloom Filter storage
  - a "Bloom Filter" is a space-efficient probabilistic data structure, that is used to test whether an
    element is a member of a set. False positive matches are possible, but false negatives are not.
    Elements can be added to the set, but not removed. Typical use cases are to avoid unnecessary
    slow disk or network access if possible, when a lot of items are involved.
    - memory use is very low, when compared to storage of all values: fewer than 10 bits per element
      are required for a 1% false positive probability, independent of the size or number of elements in
      the set - for instance, storing 10,000,000 items presence with 1% of false positive ratio would
      consume only 11.5 MB of memory, using 7 hash functions
    - use Insert() methods to add an item to the internal bits array, and Reset() to clear all bits array, if
      needed
    - MayExist() function would check if the supplied item was probably set
    - SaveTo() and LoadFrom() methods allow transmission of the bits array, for a disk/database
      storage or transmission over a network
    - internally, several (hardware-accelerated) crc32c hash functions will be used, with some random
      seed values, to simulate several hashing functions
    - Insert/MayExist/Reset methods are thread-safe

constructor Create(const aSaved: RawByteString; aMagic: cardinal=$B1003F11); reintroduce; overload;
  Initialize the internal bits storage from a SaveTo() binary buffer
  - this constructor will initialize the internal bits array calling LoadFrom()
constructor Create(aSize: integer; aFalsePositivePercent: double = 1); reintroduce; overload;

Initialize the internal bits storage for a given number of items
- by default, internal bits array size will be guessed from a 1% false positive rate - but you may specify another value, to reduce memory use
- this constructor would compute and initialize Bits and HashFunctions corresponding to the expected false positive ratio

function LoadFrom(const aSaved: RawByteString; aMagic: cardinal=$B1003F11): boolean; overload;

Read the internal bits array from a binary buffer
- as previously serialized by the SaveTo method
- may be used to transmit or store the state of a dataset

function LoadFrom(P: PByte; PLen: integer; aMagic: cardinal=$B1003F11): boolean; overload; virtual;

Read the internal bits array from a binary buffer
- as previously serialized by the SaveTo method
- may be used to transmit or store the state of a dataset

function MayExist(aValue: pointer; aValueLen: integer): boolean; overload;

Returns TRUE if the supplied items was probably set via Insert()
- some false positive may occur, but not much than FalsePositivePercent
- this method is thread-safe

function MayExist(const aValue: RawByteString): boolean; overload;

Returns TRUE if the supplied items was probably set via Insert()
- some false positive may occur, but not much than FalsePositivePercent
- this method is thread-safe

function SaveTo(aMagic: cardinal=$B1003F11): RawByteString; overload;

Store the internal bits array into a binary buffer
- may be used to transmit or store the state of a dataset, avoiding to recompute all Insert() at program startup, or to synchronize networks nodes information and reduce the number of remote requests

procedure Insert(aValue: pointer; aValueLen: integer); overload; virtual;

Add an item in the internal bits array storage
- this method is thread-safe

procedure Insert(const aValue: RawByteString); overload;

Add an item in the internal bits array storage
- this method is thread-safe

procedure Reset; virtual;

Clear the internal bits array storage
- you may call this method after some time, if some items may have been removed, to reduce false positives
- this method is thread-safe
procedure SaveTo(aDest: TFileBufferWriter; aMagic: cardinal=$B1003F11); overload;

  Store the internal bits array into a binary buffer
  - may be used to transmit or store the state of a dataset, avoiding to recompute all Insert() at program startup, or to synchronize networks nodes information and reduce the number of remote requests

property Bits: cardinal read fBits;
  Number of bits stored in the internal bits array

property FalsePositivePercent: double read fFalsePositivePercent;
  Expected percentage (1..100) of false positive results for MayExists()

property HashFunctions: cardinal read fHashFunctions;
  How many hash functions would be applied for each Insert()

property Inserted: cardinal read GetInserted;
  How many times the Insert() method has been called

property Size: cardinal read fSize;
  Maximum number of items which are expected to be inserted

TSynBloomFilterDiff = class(TSynBloomFilter)
  Implements a thread-safe differential Bloom Filter storage
  - this inherited class is able to compute incremental serialization of its internal bits array, to reduce network use
  - an obfuscated revision counter is used to identify storage history

function DiffKnownRevision(const aDiff: RawByteString): Int64;
  Retrieve the revision number from an incremental binary buffer
  - returns 0 if the supplied binary buffer does not match this bloom filter

function LoadFromDiff(const aDiff: RawByteString): boolean;
  Read the internal bits array from an incremental binary buffer
  - as previously serialized by the SaveToDiff() method
  - may be used to transmit or store the state of a dataset
  - returns false if the supplied content is incorrect, e.g. if the known revision is deprecated

function SaveToDiff(const aKnownRevision: Int64): RawByteString;
  Store the internal bits array into an incremental binary buffer
  - here the difference from a previous SaveToDiff revision will be computed
  - if aKnownRevision is outdated (e.g. if equals 0), the whole bits array would be returned, and around 10 bits per item would be transmitted (for 1% false positive ratio)
  - incremental retrieval would then return around 10 bytes per newly added item since the last snapshot reference state (with 1% ratio, i.e. 7 hash functions)

procedure DiffSnapshot;
  Use the current internal bits array state as known revision
  - is done the first time SaveToDiff() is called, then after 1/32th of the filter size has been inserted (see SnapshotAfterInsertCount property), or after SnapShotAfterMinutes property timeout period

SynTable.pas unit - Rev. 1.18 Page 1750 of 2552
procedure Insert(aValue: pointer; aValueLen: integer); override;
   Add an item in the internal bits array storage
   - this overloaded thread-safe method would compute fRevision

procedure Reset; override;
   Clear the internal bits array storage
   - this overloaded thread-safe method would reset fRevision

property Revision: Int64 read fRevision;
   The opaque revision number of this internal storage
   - is in fact the Unix timestamp shifted by 31 bits, and an incremental counter: this pattern will
     allow consistent IDs over several ServPanels

property SnapshotAfterInsertCount: cardinal read fSnapshotAfterInsertCount write fSnapshotAfterInsertCount;
   After how many Insert() the internal bits array storage should be promoted as known revision
   - equals Size div 32 by default

property SnapShotAfterMinutes: cardinal read fSnapShotAfterMinutes write fSnapShotAfterMinutes;
   After how many time the internal bits array storage should be promoted as known revision
   - equals 30 minutes by default

TSynCache = class(TSynPersistentLock)
   Implement a cache of some key/value pairs, e.g. to improve reading speed
   - used e.g. by TSQLDataBase for caching the SELECT statements results in an internal JSON format
     (which is faster than a query to the SQLite3 engine)
   - internally make use of an efficient hashing algorithm for fast response (i.e. TSynNameValue will
     use the TDynArrayHashed wrapper mechanism)
   - this class is thread-safe if you use properly the associated Safe lock

constructor Create(aMaxCacheRamUsed: cardinal=16 shl 20; aCaseSensitive: boolean=false; aTimeoutSeconds: cardinal=0); reintroduce;
   Initialize the internal storage
   - aMaxCacheRamUsed can set the maximum RAM to be used for values, in bytes (default is 16 MB), after which the cache is flushed
   - by default, key search is done case-insensitively, but you can specify another option here
   - by default, there is no timeout period, but you may specify a number of seconds of inactivity
     (i.e. no Add call) after which the cache is flushed

function AddOrUpdate(const aKey, aValue: RawUTF8; aTag: PtrInt): boolean;
   Add a Key/Value pair in the cache entries
   - returns true if aKey was not existing yet, and aValue has been stored
   - returns false if aKey did already exist in the internal cache, and its entry has been updated with
     the supplied aValue/aTag
   - this method is thread-safe, using the Safe locker of this instance

function Count: integer;
   Number of entries in the cache
function Find(const aKey: RawUTF8; aResultTag: PPtrInt=nil): RawUTF8;

Find a Key in the cache entries
- return " if nothing found: you may call Add() just after to insert the expected value in the cache
- return the associated Value otherwise, and the associated integer tag if aResultTag address is supplied
- this method is not thread-safe, unless you call Safe.Lock before calling Find(), and Safe.Unlock after calling Add()

function Reset: boolean;

Called after a write access to the database to flush the cache
- set Count to 0
- release all cache memory
- returns TRUE if was flushed, i.e. if there was something in cache
- this method is thread-safe, using the Safe locker of this instance

procedure Add(const aValue: RawUTF8; aTag: PtrInt);

Add a Key and its associated value (and tag) to the cache entries
- you MUST always call Find() with the associated Key first
- this method is not thread-safe, unless you call Safe.Lock before calling Find(), and Safe.Unlock after calling Add()

property MaxRamUsed: cardinal read fMaxRamUsed;

The maximum RAM to be used for values, in bytes
- the cache is flushed when ValueSize reaches this limit
- default is 16 MB (16 shl 20)

property RamUsed: cardinal read fRamUsed;

The current global size of Values in RAM cache, in bytes

property Safe: PSynLocker read fSafe;

Access to the internal locker, for thread-safe process
- Find/Add methods calls should be protected as such:
  cache.Safe.Lock;
  try
    ... cache.Find/cache.Add ...
  finally
    cache.Safe.Unlock;
  end;

property TimeoutSeconds: cardinal read fTimeoutSeconds;

After how many seconds between Add() calls the cache should be flushed
- equals 0 by default, meaning no time out

TSynQueue = class(TSynPersistentLock)

Thread-safe FIFO (First-In-First-Out) in-order queue of records
- uses internally a dynamic array storage, with a sliding algorithm (more efficient than the FPC or Delphi TQueue)
constructor Create(aTypeInfo: pointer); reintroduce; virtual;

Initialize the queue storage
- aTypeInfo should be a dynamic array TypeInfo() RTTI pointer, which would store the values within this TSynQueue instance

destructor Destroy; override;

Finalize the storage
- would release all internal stored values, and call WaitPopFinalize

function Capacity: integer;

Returns how much slots is currently reserved in memory
- the queue has an optimized auto-sizing algorithm, you can use this method to return its current capacity
- this method is thread-safe

function Count: Integer;

Returns how many items are currently stored in this queue
- this method is thread-safe

function Peek(out aValue): boolean;

Lookup one item from the queue, as FIFO (First-In-First-Out)
- returns true if aValue has been filled with a pending item, without removing it from the queue (as Pop method does)
- returns false if the queue is empty
- this method is thread-safe, since it will lock the instance

function Pending: boolean;

Returns true if there are some items currently pending in the queue
- slightly faster than checking Count=0, and much faster than Pop or Peek

function Pop(out aValue): boolean;

Extract one item from the queue, as FIFO (First-In-First-Out)
- returns true if aValue has been filled with a pending item, which is removed from the queue (use Peek if you don’t want to remove it)
- returns false if the queue is empty
- this method is thread-safe, since it will lock the instance

function PopEquals(aAnother: pointer; aCompare: TDynArraySortCompare; out aValue): boolean;

Extract one matching item from the queue, as FIFO (First-In-First-Out)
- the current pending item is compared with aAnother value

function WaitPeekLocked(aTimeoutMS: integer; const aWhenIdle: TThreadMethod): pointer;

Waiting lookup of one item from the queue, as FIFO (First-In-First-Out)
- returns a pointer to a pending item within the specified aTimeoutMS time - the Safe.Lock is still there, so that caller could check its content, then call Pop() if it is the expected one, and eventually always call Safe.Unlock
- returns nil if nothing was pushed into the queue in time
- this method is thread-safe, but will lock the instance only if needed
```pascal
function WaitPop(aTimeoutMS: integer; const aWhenIdle: TThreadMethod; out aValue; aCompared: pointer=nil; aCompare: TDynArraySortCompare=nil): boolean;

Waiting extract of one item from the queue, as FIFO (First-In-First-Out)
- returns true if aValue has been filled with a pending item within the specified aTimeoutMS time
- returns false if nothing was pushed into the queue in time, or if WaitPopFinalize has been called
- aWhenIdle could be assigned e.g. to VCL/LCL Application.ProcessMessages
- you can optionally compare the pending item before returning it (could be used e.g. when several threads are putting items into the queue)
- this method is thread-safe, but will lock the instance only if needed

procedure Clear;

Delete all items currently stored in this queue, and void its capacity
- this method is thread-safe, since it will lock the instance

procedure Push(const aValue);

Store one item into the queue
- this method is thread-safe, since it will lock the instance

procedure Save(out aDynArrayValues; aDynArray: PDynArray=nil);

Initialize a dynamic array with the stored queue items
- aDynArrayValues should be a variable defined as aTypeInfo from Create
- you can retrieve an optional TDynArray wrapper, e.g. for binary or JSON persistence
- this method is thread-safe, and will make a copy of the queue data

procedure WaitPopFinalize(aTimeoutMS: integer=100);

Ensure any pending or future WaitPop() returns immediately as false
- is always called by Destroy destructor
- could be also called e.g. from an UI OnClose event to avoid any lock
- this method is thread-safe, but will lock the instance only if needed

TObjectListSorted = class(TSynPersistentLock)

Maintain a thread-safe sorted list of TSynPersistentLock objects
- will use fast O(\log(n)) binary search for efficient search - it is a lighter alternative to TObjectListHashedAbstract/TObjectListPropertyHashed if hashing has a performance cost (e.g. if there are a few items, or deletion occurs regularly)
- in practice, insertion becomes slower after around 100,000 items stored
- expect to store only TSynPersistentLock inherited items, so that the process is explicitly thread-safe
- inherited classes should override the Compare and NewItem abstract methods

destructor Destroy; override;

Finalize the list

function Delete(const Value): boolean;

Remove a given TSynPersistentLock instance from a value
```
function FindLocked(const Value): pointer;

Search a given TSynPersistentLock instance from a value
- if returns not nil, caller should make result.Safe.UnLock once finished
- will use the TObjectListSortedCompare function for the search

function FindOrAddLocked(const Value; out added: boolean): pointer;

Search or add a given TSynPersistentLock instance from a value
- if returns not nil, caller should make result.Safe.UnLock once finished
- added is TRUE if a new void item has just been created
- will use the TObjectListSortedCompare function for the search

property Count: Integer read fCount;

How many items are actually stored

property ObjArray: TSynPersistentLockDynArray read fObjArray;

Low-level access to the stored items
- warning: use should be protected by Lock.Enter/Lock.Leave

TSynPersistentStore = class(TSynPersistentLock)

Abstract high-level handling of (SynLZ-)compressed persisted storage
- LoadFromReader/SaveToWriter abstract methods should be overriden with proper binary persistence implementation

constructor Create(const aName: RawUTF8); reintroduce; overload; virtual;

Initialize a void storage with the supplied name

constructor CreateFrom(const aBuffer: RawByteString; aLoad: TAlgoCompressLoad = aclNormal);

Initialize a storage from a SaveTo persisted buffer
- raise a EFastReader exception on decoding error

constructor CreateFromBuffer(aBuffer: pointer; aBufferLen: integer; aLoad: TAlgoCompressLoad = aclNormal);

Initialize a storage from a SaveTo persisted buffer
- raise a EFastReader exception on decoding error

constructor CreateFromFile(const aFileName: TFileName; aLoad: TAlgoCompressLoad = aclNormal);

Initialize a storage from a SaveTo persisted buffer
- raise a EFastReader exception on decoding error

function LoadFromFile(const aFileName: TFileName; aLoad: TAlgoCompressLoad = aclNormal): boolean;

Initialize the storage from a SaveToFile content
- actually call the LoadFromReader() virtual method for persistence
- returns false if the file is not found, true if the file was loaded without any problem, or raise a EFastReader exception on decoding error

function SaveTo(nocompression: boolean=false; BufLen: integer=65536; ForcedAlgo: TAlgoCompress=nil; BufferOffset: integer=0): RawByteString; overload;

Persist the content as a SynLZ-compressed binary blob
- just an overloaded wrapper
function SaveToFile(const aFileName: TFileName; nocompression: boolean=false; BufLen: integer=65536; ForcedAlgo: TAlgoCompress=nil): PtrUInt;

Persist the content as a SynLZ-compressed binary file
- to be retrieved later on via LoadFromFile method
- returns the number of bytes of the resulting file
- actually call the SaveTo method for persistence

procedure LoadFrom(const aBuffer: RawByteString; aLoad: TAlgoCompressLoad = aclNormal); overload;

Fill the storage from a SaveTo persisted buffer
- actually call the LoadFromReader() virtual method for persistence
- raise a EFastReader exception on decoding error

procedure LoadFrom(aBuffer: pointer; aBufferLen: integer; aLoad: TAlgoCompressLoad = aclNormal); virtual;

Initialize the storage from a SaveTo persisted buffer
- actually call the LoadFromReader() virtual method for persistence
- raise a EFastReader exception on decoding error

procedure SaveTo(out aBuffer: RawByteString; nocompression: boolean=false; BufLen: integer=65536; ForcedAlgo: TAlgoCompress=nil; BufferOffset: integer=0); overload;

Persist the content as a SynLZ-compressed binary blob
- to be retrieved later on via LoadFrom method
- actually call the SaveToWriter() protected virtual method for persistence
- you can specify ForcedAlgo if you want to override the default AlgoSynLZ
- BufferOffset could be set to reserve some bytes before the compressed buffer

property LoadFromLastUncompressed: integer read fLoadFromLastUncompressed;
After a LoadFrom(), contains the uncompressed data size read

property Name: RawUTF8 read fName;
One optional text associated with this storage
- you can define this field as published to serialize its value in log/JSON

property SaveToLastUncompressed: integer read fSaveToLastUncompressed;
After a SaveTo(), contains the uncompressed data size written

TSynPersistentStoreJson = class(TSynPersistentStore)
Implement binary persistence and JSON serialization (not deserialization)

function SaveToJSON(reformat: TTextWriterJSONFormat = jsonCompact): RawUTF8;
Serialize this instance as a JSON object

TRawByteStringGroupValue = record
Item as stored in a TRawByteStringGroup instance

TRawByteStringGroup = object(TObject)
Store several RawByteString content with optional concatenation
Count: integer;
   How many items are currently stored in Values[]

LastFind: integer;
   Naive but efficient cache for Find()

Position: integer;
   The current size of data stored in Values[]

Values: TRawByteStringGroupValueDynArray;
   Actual list storing the data

function AsBytes: TByteDynArray;
   Return all content as a single TByteDynArray

function AsText: RawByteString;
   Return all content as a single RawByteString
   - will also compact the Values[] array into a single item (which is returned)

function Equals(const aAnother: TRawByteStringGroup): boolean;
   Compare two TRawByteStringGroup instance stored text

function Find(aPosition: integer): PRawByteStringGroupValue; overload;
   Returns a pointer to Values[] containing a given position
   - returns nil if not found

function Find(aPosition, aLength: integer): pointer; overload;
   Returns a pointer to Values[].Value containing a given position and length
   - returns nil if not found

function FindAsText(aPosition, aLength: integer): RawByteString; overload;
   Returns the text at a given position in Values[]
   - text should be in a single Values[] entry

procedure Add(const aItem: RawByteString); overload;
   Add a new item to Values[]

procedure Add(const aAnother: TRawByteStringGroup); overload;
   Add another TRawByteStringGroup to Values[]

procedure Add(aItem: pointer; aItemLen: integer); overload;
   Add a new item to Values[]

procedure AddFromReader(var aReader: TFastReader);
   Add another TRawByteStringGroup previously serialized via WriteString()

procedure AppendTextAndClear(var aDest: RawByteString);
   Append stored information into another RawByteString, and clear content

procedure Clear;
   Clear any stored information

procedure Compact;
   Compact the Values[] array into a single item
   - is also used by AsText to compute a single RawByteString
procedure FindAsText(aPosition, aLength: integer; out aText: RawByteString);
overload;

*Returns the text at a given position in Values[]*
- text should be in a single Values[] entry

procedure FindAsVariant(aPosition, aLength: integer; out aDest: variant);

*Returns the text at a given position in Values[]*
- text should be in a single Values[] entry
- explicitly returns null if the supplied text was not found

procedure FindMove(aPosition, aLength: integer; aDest: pointer);

*Copy the text at a given position in Values[]*
- text should be in a single Values[] entry

procedure FindWrite(aPosition, aLength: integer; W: TTextWriter; Escape:
TTextWriterKind=twJSONEscape; TrailingCharsToIgnore: integer=0);

*Append the text at a given position in Values[], JSON escaped by default*
- text should be in a single Values[] entry

procedure FindWriteBase64(aPosition, aLength: integer; W: TTextWriter; withMagic:
boolean);

*Append the blob at a given position in Values[], base-64 encoded*
- text should be in a single Values[] entry

procedure RemoveLastAdd;

*Low-level method to abort the latest Add() call*
- warning: will work only once, if an Add() has actually been just called: otherwise, the behavior is unexpected, and may wrongly truncate data

procedure Write(W: TTextWriter; Escape: TTextWriterKind=twJSONEscape); overload;

*Save all content into a TTextWriter instance*

procedure WriteBinary(W: TFileBufferWriter); overload;

*Save all content into a TFileBufferWriter instance*

procedure WriteString(W: TFileBufferWriter);

*Save all content as a string into a TFileBufferWriter instance*
- storing the length as WriteVarUInt32() prefix

TPropNameList = object(TObject)

*Simple stack-allocated type for handling a non-void type names list*
- Delphi "object" is buggy on stack -> also defined as record with methods

Count: Integer;

*How many items are currently in Values[]*

Values: TRawUTF8DynArray;

*The actual names storage*

function AddPropName(const Value: RawUTF8): Boolean;

*If Value is in Values[0..Count-1] using IdemPropNameU() returns FALSE*
- otherwise, returns TRUE and add Value to Values[]
- any Value="" is rejected
function FindPropName(const Value: RawUTF8): Integer;
Search for a Value within Values[0..Count-1] using IdemPropNameU()

procedure Init;
Initialize the list
- set Count := 0

TSynUniqueIdentifierBits = object(TObject)
Map 64-bit integer unique identifier internal memory structure
- as stored in TSynUniqueIdentifier = Int64 values, and computed by
TSynUniqueIdentifierGenerator
- bits 0..14 map a 15-bit increasing counter (collision-free)
- bits 15..30 map a 16-bit process identifier
- bits 31..63 map a 33-bit UTC time, encoded as seconds since Unix epoch

Value: TSynUniqueIdentifier;
The actual 64-bit storage value
- in practice, only first 63 bits are used

function AsVariant: variant;
Convert this identifier as an explicit TDocVariant JSON object
- returns e.g.
  
  {"Created":"2016-04-19T15:27:58","Identifier":1,"Counter":1,
  "Value":3137644716930138113,"Hex":"2B8B273F00008001"}

function Counter: word;
15-bit counter (0..32767), starting with a random value

function CreateDateTime: TDateTime;
Extract the UTC generation timestamp from the identifier as TDateTime
- time is expressed in Coordinated Universal Time (UTC), not local time

function CreateTimeLog: TTi
timeLog;
Extract the UTC generation timestamp from the identifier
- time is expressed in Coordinated Universal Time (UTC), not local time

function CreateTimeUnix: TUnixTime;
Low-endian 4-byte value representing the seconds since the Unix epoch
- time is expressed in Coordinated Universal Time (UTC), not local time
- it uses in fact a 33-bit resolution, so is "Year 2038" bug-free

function Equal(const Another: TSynUniqueIdentifierBits): boolean;
Compare two Identifiers

function FromHexa(const hexa: RawUTF8): boolean;
Fill this unique identifier back from a 16 chars hexadecimal string
- returns TRUE if the supplied hexadecimal is on the expected format
- returns FALSE if the supplied text is invalid
function ProcessID: TSynUniqueIdentifierProcess;
  // 16-bit unique process identifier
  // as specified to TSynUniqueIdentifierGenerator constructor

function ToHexa: RawUTF8;
  // Convert the identifier into a 16 chars hexadecimal string

procedure From(const AID: TSynUniqueIdentifier);
  // Fill this unique identifier structure from its TSynUniqueIdentifier value
  // - is just a wrapper around PInt64(@self)^

procedure FromDateTime(const aDateTime: TDateTime);
  // Fill this unique identifier with a fake value corresponding to a given timestamp
  // - may be used e.g. to limit database queries on a particular time range
  // - bits 0..30 would be 0, i.e. would set Counter = 0 and ProcessID = 0

procedure FromUnixTime(const aUnixTime: TUnixTime);
  // Fill this unique identifier with a fake value corresponding to a given timestamp
  // - may be used e.g. to limit database queries on a particular time range
  // - bits 0..30 would be 0, i.e. would set Counter = 0 and ProcessID = 0

procedure ToVariant(out result: variant);
  // Convert this identifier to an explicit TDocVariant JSON object
  // returns e.g.
  // {"Created":"2016-04-19T15:27:58","Identifier":1,"Counter":1,
  //  "Value":3137644716930138113,"Hex":"2B8B273F00008001"}

TSynUniqueIdentifierGenerator = class(TSynPersistent)
  // Thread-safe 64-bit integer unique identifier computation
  // - may be used on client side for something similar to a MongoDB ObjectId, but compatible with
  // TSQLRecord.ID: TID properties, since it will contain a 63-bit unsigned integer, following our ORM
  // expectations
  // - each identifier would contain a 16-bit process identifier, which is supplied by the application, and
  // should be unique for this process at a given time
  // - identifiers may be obfuscated as hexadecimal text, using both encryption and digital signature

constructor Create(aIdentifier: TSynUniqueIdentifierProcess; const aSharedObfuscationKey: RawUTF8=''); reintroduce;
  // Initialize the generator for the given 16-bit process identifier
  // - you can supply an obfuscation key, which should be shared for the whole system, so that you may
  // use FromObfuscated/ToObfuscated methods

destructor Destroy; override;
  // Finalize the generator structure

function ComputeNew: Int64; overload;
  // Return a new unique ID, type-casted to an Int64
function FromObfuscated(const aObfuscated: TSynUniqueIdentifierObfuscated; out aIdentifier: TSynUniqueIdentifier): boolean;

Retrieve a TSynUniqueIdentifier from 24 chars cyphered hexadecimal text
- any file extension (e.g. '.jpeg') would be first deleted from the supplied obfuscated text
- returns true if the supplied obfuscated text has the expected layout and a valid digital signature
- returns false if the supplied obfuscated text is invalid

function ToObfuscated(const aIdentifier: TSynUniqueIdentifier): TSynUniqueIdentifierObfuscated;

Map a TSynUniqueIdentifier as 24 chars cyphered hexadecimal text
- cyphering includes simple key-based encryption and a CRC-32 digital signature

procedure ComputeFromDateTime(const aDateTime: TDateTime; out result: TSynUniqueIdentifierBits);

Return an unique ID matching this generator pattern, at a given timestamp
- may be used e.g. to limit database queries on a particular time range

procedure ComputeFromUnixTime(const aUnixTime: TUnixTime; out result: TSynUniqueIdentifierBits);

Return an unique ID matching this generator pattern, at a given timestamp
- may be used e.g. to limit database queries on a particular time range

procedure ComputeNew(out result: TSynUniqueIdentifierBits); overload;

Return a new unique ID
- this method is very optimized, and would use very little CPU

property ComputedCount: Int64 read GetComputedCount;

How many times ComputeNew method has been called

property CryptoCRC: cardinal read fCryptoCRC;

Some 32-bit value, derivated from aSharedObfuscationKey as supplied to the class constructor
- FromObfuscated and ToObfuscated methods will validate their hexadecimal content with this value to secure the associated CRC
- may be used e.g. as system-depending salt

property Identifier: TSynUniqueIdentifierProcess read fIdentifier;

The process identifier, associated with this generator

property Safe: TSynLocker read fSafe;

Direct access to the associated mutex

TSynPersistentWithPassword = class(TSynPersistent)

Abstract TSynPersistent class allowing safe storage of a password
- the associated Password, e.g. for storage or transmission encryption will be persisted encrypted with a private key (which can be customized)
- if default simple symmetric encryption is not enough, you may define a custom TSynPersistentWithPasswordUserCrypt callback, e.g. to Syncrypto's CryptDataForCurrentUser, for hardened password storage
- a published property should be defined as such in inherited class:
  property PasswordPropertyName: RawUTF8 read fPassword write fPassword;
- use the PassWordPlain property to access to its uncyphered value
```pascal
destructor Destroy; override;

Finalize the instance

class function ComputePassword(PlainPassword: pointer; PlainPasswordLen: integer;
CustomKey: cardinal=0): RawUTF8; overload;

This class method could be used to compute the encrypted password from a binary digest, ready
to be stored as JSON, according to a given private key
  - just a wrapper around ComputePassword(BinToBase64URI())

class function ComputePassword(const PlainPassword: RawUTF8; CustomKey:
cardinal=0): RawUTF8; overload;

This class method could be used to compute the encrypted password, ready to be stored as JSON,
according to a given private key

class function ComputePlainPassword(const CypheredPassword: RawUTF8; CustomKey:
cardinal=0; const AppSecret: RawUTF8=''): RawUTF8;

This class method could be used to decrypt a password, stored as JSON, according to a given
private key
  - may trigger a ESynException if the password was stored using a custom
    TSynPersistentWithPasswordUserCrypt callback, and the current user doesn't match the
    expected user stored in the field

function GetPasswordFieldAddress: pointer;

Low-level function used to identify if a given field is a Password
  - this method is used e.g. by TJSONSerializer.WriteObject to identify the password field, since its
    published name is set by the inherited classes

property Key: cardinal read GetKey write fKey;

The private key used to cypher the password storage on serialization
  - application can override the default 0 value at runtime

property PasswordPlain: RawUTF8 read GetPasswordPlain write SetPassWordPlain;

Access to the associated unencrypted Password value
  - read may trigger a ESynException if the password was stored using a custom
    TSynPersistentWithPasswordUserCrypt callback, and the current user doesn't match the
    expected user stored in the field

tsynUserPassword = class(TSynPersistentWithPassword)

Could be used to store a credential pair, as user name and password
  - password will be stored with TSynPersistentWithPassword encryption

property Password: RawUTF8 read FPassword write FPassword;

The associated encrypted password
  - use the PasswordPlain public property to access to the unencrypted password

property UserName: RawUTF8 read FUserName write FUserName;

The associated user name
```
TSynConnectionDefinition = class(TSynPersistentWithPassword)

Handle safe storage of any connection properties
- would be used by SynDB.pas to serialize TSQlDBCConnectionProperties, or by mORMot.pas to serialize TSQLRest instances
- the password will be stored as Base64, after a simple encryption as defined by TSynPersistentWithPassword
- typical content could be:

```json
{
    "Kind": "TSQLDBSQLite3ConnectionProperties",
    "ServerName": "server",
    "DatabaseName": "",
    "User": "",
    "Password": "PtvlPA=="
}
```

- the "Kind" value will be used to let the corresponding TSQLRest or TSQlDBCConnectionProperties NewInstance*() class methods create the actual instance, from its class name

constructor CreateFromJSON(const JSON: RawUTF8; Key: cardinal=0); virtual;

Unserialize the database definition from JSON
- as previously serialized with the SaveToJSON method
- you can specify a custom Key used for password encryption, if the default value is not safe enough for you
- this method won't use JSONToObject() so avoid any dependency to mORMot.pas

function SaveToJSON: RawUTF8; virtual;

Serialize the database definition as JSON
- this method won't use ObjectToJSON() so avoid any dependency to mORMot.pas

property DatabaseName: RawUTF8 read fDatabaseName write fDatabaseName;
The associated database name (if any), or additional options

property Kind: string read fKind write fKind;
The class name implementing the connection or TSQLRest instance
- will be used to instantiate the expected class type

property Password: RawUTF8 read fPassword write fPassword;
The associated Password, e.g. for storage or transmission encryption
- will be persisted encrypted with a private key
- use the PassWordPlain property to access to its uncyphered value

property ServerName: RawUTF8 read fServerName write fServerName;
The associated server name (or file, for SQLite3) to be connected to

property User: RawUTF8 read fUser write fUser;
The associated User Identifier (if any)

TSynAuthenticationAbstract = class(TObject)

Abstract authentication class, implementing safe token/challenge security and a list of active sessions
- do not use this class, but plain TSynAuthentication
constructor Create;
    Initialize the authentication scheme

destructor Destroy; override;
    Finalize the authentication

class function ComputeHash(Token: Int64; const UserName, PassWord: RawUTF8): cardinal; virtual;
    To be used to compute a Hash on the client side, for a given Token
    - the token should have been retrieved from the server, and the client should compute and
    return this hash value, to perform the authentication challenge and create the session
    - internal algorithm is not cryptographic secure, but fast and safe

function CreateSession(const User: RawUTF8; Hash: cardinal): integer; virtual;
    Create a new session
    - should return 0 on authentication error, or an integer session ID
    - this method will check the User name and password, and create a new session

function CurrentToken: Int64;
    Returns the current identification token
    - to be sent to the client for its authentication challenge

function SessionExists(aID: integer): boolean;
    Check if the session exists in the internal list

procedure AuthenticateUser(const aName, aPassword: RawUTF8); virtual;
    Register one credential for a given user
    - this abstract method will raise an exception: inherited classes should implement them as
    expected

procedure DisauthenticateUser(const aName: RawUTF8); virtual;
    Unregister one credential for a given user
    - this abstract method will raise an exception: inherited classes should implement them as
    expected

procedure RemoveSession(aID: integer);
    Delete a session

property SessionsCount: integer read fSessionsCount;
    The number of current opened sessions

property UsersCount: integer read GetUsersCount;
    The number of registered users

TSynAuthentication = class(TSynAuthenticationAbstract)
    Simple authentication class, implementing safe token/challenge security
    - maintain a list of user / name credential pairs, and a list of sessions
    - is not meant to handle authorization, just plain user access validation
    - used e.g. by TSQDBConnection.RemoteProcessMessage (on server side) and
    TSQLDProxyConnectionPropertiesAbstract (on client side) in SynDB.pas
constructor Create(const aUserName: RawUTF8=''; const aPassword: RawUTF8=''); reintroduce;
   Initialize the authentication scheme
   - you can optionally register one user credential

procedure AuthenticateUser(const aName, aPassword: RawUTF8); override;
   Register one credential for a given user

procedure DisauthenticateUser(const aName: RawUTF8); override;
   Unregister one credential for a given user

TIPBan = class(TSynPersistentStore)
   Optimized thread-safe storage of a list of IP v4 addresses
   - can be used e.g. as white-list or black-list of clients
   - will maintain internally a sorted list of 32-bit integers for fast lookup
   - with optional binary persistence

function Add(const aIP: RawUTF8): boolean;
   Register one IP to the list

function Delete(const aIP: RawUTF8): boolean;
   Unregister one IP to the list

function DynArrayLocked: TDynArray;
   Creates a TDynArray wrapper around the stored list of values
   - could be used e.g. for binary persistence
   - warning: caller should make Safe.Unlock when finished

function Exists(const aIP: RawUTF8): boolean;
   Returns true if the IP is in the list

property Count: integer read fCount;
   How many IPs are currently banned

property IP4: TIntegerDynArray read fIP4;
   Low-level access to the internal IPv4 list
   - 32-bit unsigned values are sorted, for fast O(log(n)) binary search

EExprParser = class(ESynException)
   Exception type used by TExprParser

TExprNode = class(TSynPersistent)
   Stores an expression search engine node, as used by TExprParser

constructor Create(nodeType: TExprNodeType); reintroduce;
   Initialize a node for the search engine

destructor Destroy; override;
   Recursively destroys the linked list of nodes (i.e. Next)

function Last: TExprNode;
   Browse all nodes until Next = nil
property Next: TExprNode read fNext;
   Points to the next node in the parsed tree

property NodeType: TExprNodeType read fNodeType;
   What is actually stored in this node

TExprNodeWordAbstract = class(TExprNode)
Abstract class to handle word search, as used by TExprParser

constructor Create(aOwner: TParserAbstract; const aWord: RawUTF8); reintroduce;
   You should override this virtual constructor for proper initialization

TParserAbstract = class(TSynPersistent)
Parent class of TExprParserAbstract

constructor Create; override;
   Initialize an expression parser

destructor Destroy; override;
   Finalize the expression parser

function Parse(const aExpression: RawUTF8): TExprParserResult;
   Initialize the parser from a given text expression

class function ParseError(const aExpression: RawUTF8): RawUTF8;
   Try this parser class on a given text expression
   - returns '' on success, or an explicit error message (e.g. 'Missing parenthesis')

property Expression: RawUTF8 read fExpression;
   The associated text expression used to define the search

property WordCount: integer read fWordCount;
   How many words did appear in the search expression

TExprParserAbstract = class(TParserAbstract)
Abstract class to parse a text expression into nodes
   you should inherit this class to provide actual text search
   - searched expressions can use parenthesis and &=AND -=WITHOUT +=OR operators, e.g. '((w1 & w2) - w3) + w4' means ((w1 and w2) without w3) or w4
   - no operator is handled like a AND, e.g. 'w1 w2' = 'w1 & w2'

TExprParserMatch = class(TExprParserAbstract)
Search expression engine using TMatch for the actual word searches

constructor Create(aCaseSensitive: boolean = true); reintroduce;
   Initialize the search engine

function Search(const aText: RawUTF8): boolean; overload;
   Returns TRUE if the expression is within the text buffer
function Search(aText: PUTF8Char; aTextLen: PtrInt): boolean; overload;

    Returns TRUE if the expression is within the text buffer

TPendingTaskListItem = packed record
Internal item definition, used by TPendingTaskList storage

    Task: RawByteString;
    The associated task, stored by representation as raw binary

    Timestamp: Int64;
    The task should be executed when TPendingTaskList.GetTimestamp reaches this value

TPendingTaskList = class(TSynPersistentLock)
Handle a list of tasks, stored as RawByteString, with a time stamp
- internal time stamps would be GetTickCount64 by default, so have a resolution of about 16 ms under Windows
- you can add tasks to the internal list, to be executed after a given delay, using a post/peek like algorithm
- execution delays are not expected to be accurate, but are best guess, according to NextTask call
- this implementation is thread-safe, thanks to the Safe internal locker

constructor Create; override;
    Initialize the list memory and resources

function NextPendingTask: RawByteString; virtual;
    Retrieve the next pending task
    - returns " if there is no scheduled task available at the current time
    - returns the next stack as defined corresponding to its specified delay

procedure AddTask(aMilliSecondsDelayFromNow: integer; const aTask: RawByteString); virtual;
    Append a task, specifying a delay in milliseconds from current time

procedure AddTasks(const aMilliSecondsDelays: array of integer; const aTasks: array of RawByteString);
    Append several tasks, specifying a delay in milliseconds between tasks
    - first supplied delay would be computed from the current time, then it would specify how much time to wait between the next supplied task

procedure Clear; virtual;
    Flush all pending tasks

property Count: integer read GetCount;
    How many pending tasks are currently defined
**property** Task: TPendingTaskListItemDynArray read fTask;

*Direct low-level access to the internal task list*
- warning: this dynamic array length is the list capacity: use Count property to retrieve the exact number of stored items
- use Safe.Lock/TryLock with a try ... finally Safe.Unlock block for thread-safe access to this array
- items are stored in increasing Timestamp, i.e. the first item is the next one which would be returned by the NextPendingTask method

**property** Timestamp: Int64 read GetTimestamp;

*Access to the internal TPendingTaskListItem.Timestamp stored value*
- corresponding to the current time
- default implementation is to return GetTickCount64, with a 16 ms typical resolution under Windows

**TSynBackgroundThreadAbstract = class(TThread)**

*Abstract TThread with its own execution content*
- you should not use this class directly, but use either TSynBackgroundThreadMethodAbstract / TSynBackgroundThreadEvent / TSynBackgroundThreadMethod and provide a much more convenient callback

**constructor** Create(const aThreadName: RawUTF8; OnBeforeExecute: TNotifyThreadEvent=nil; OnAfterExecute: TNotifyThreadEvent=nil; CreateSuspended: boolean=false); reintroduce;

*Initialize the thread*
- you could define some callbacks to nest the thread execution, e.g. assigned to TSQLRestServer.BeginCurrentThread/EndCurrentThread, or at least set OnAfterExecute to TSynLogFamily.OnThreadEnded

**destructor** Destroy; override;

*Release used resources*

**procedure** Start;

*Method to be called to start the thread*
- Resume is deprecated in the newest RTL, since some OS - e.g. Linux - do not implement this pause/resume feature; we define here this method for older versions of Delphi

**procedure** Terminate; reintroduce;

*Reintroduced to call TerminatedSet*

**procedure** TerminatedSet; virtual;

*Properly terminate the thread*
- called by reintroduced Terminate

**procedure** WaitForNotExecuting(maxMS: integer=500);

*Wait for Execute/ExecuteLoop to be ended (i.e. fExecute<>exRun)*

**property** Pause: boolean read fExecuteLoopPause write SetExecuteLoopPause;

*Temporary stop the execution of ExecuteLoop, until set back to false*
- may be used e.g. by TSynBackgroundTimer to delay the process of background tasks
property ProcessEvent: TEvent read fProcessEvent;

Access to the low-level associated event used to notify task execution to the background thread
- you may call ProcessEvent.SetEvent to trigger the internal process loop

property Terminated;

Defined as public since may be used to terminate the processing methods

TSynBackgroundThreadMethodAbstract = class(TSynBackgroundThreadAbstract)

Abstract TThread able to run a method in its own execution content
- typical use is a background thread for processing data or remote access, while the UI will be still responsive by running OnIdle event in loop: see e.g. how TSQLRestClientURI.OnIdle handle this in mORMot.pas unit
- you should not use this class directly, but inherit from it and override the Process method, or use either TSynBackgroundThreadEvent / TSynBackgroundThreadMethod and provide a much more convenient callback

constructor Create(aOnIdle: TOnIdleSynBackgroundThread; const aThreadName: RawUTF8; OnBeforeExecute: TNotifyThreadEvent=nil; OnAfterExecute: TNotifyThreadEvent=nil); reintroduce;

Initialize the thread
- if aOnIdle is not set (i.e. equals nil), it will simply wait for the background process to finish until RunAndWait() will return
- you could define some callbacks to nest the thread execution, e.g. assigned to TSQLRestServer.BeginCurrentThread/EndCurrentThread

destructor Destroy; override;

Finalize the thread

function RunAndWait(OpaqueParam: pointer): boolean;

Launch Process abstract method asynchronously in the background thread
- wait until process is finished, calling OnIdle() callback in the meanwhile
- any exception raised in background thread will be translated in the caller thread
- returns false if self is not set, or if called from the same thread as it is currently processing (to avoid race condition from OnIdle() callback)
- returns true when the background process is finished
- OpaqueParam will be used to specify a thread-safe content for the background process
- this method is thread-safe, that is it will wait for any started process already launch by another thread: you may call this method from any thread, even if its main purpose is to be called from the main UI thread


Optional callback event triggered in Execute after each Process


Optional callback event triggered in Execute before each Process
property OnIdle: TOnIdleSynBackgroundThread read fOnIdle write fOnIdle;
Set a callback event to be executed in loop during remote blocking process, e.g. to refresh the UI during a somewhat long request
- you can assign a callback to this property, calling for instance Application.ProcessMessages, to execute the remote request in a background thread, but let the UI still be reactive: the TLoginForm.OnIdleProcess and OnIdleProcessForm methods of mORMotUILogin.pas will match this property expectations
- if OnIdle is not set (i.e. equals nil), it will simply wait for the background process to finish until RunAndWait() will return

property OnIdleBackgroundThreadActive: Boolean read GetOnIdleBackgroundThreadActive;
TRUE if the background thread is active, and OnIdle event is called during process
- to be used e.g. to ensure no re-entrance from User Interface messages

TSynBackgroundThreadEvent = class(TSynBackgroundThreadMethodAbstract)
Allow background thread process of a method callback

constructor Create(aOnProcess: TOnProcessSynBackgroundThread; aOnIdle: TOnIdleSynBackgroundThread; const aThreadName: RawUTF8); reintroduce;
Initialize the thread
- if aOnIdle is not set (i.e. equals nil), it will simply wait for the background process to finish until RunAndWait() will return

Provide a method handler to be execute in the background thread
- triggered by RunAndWait() method - which will wait until finished
- the OpaqueParam as specified to RunAndWait() will be supplied here

TSynBackgroundThreadMethod = class(TSynBackgroundThreadMethodAbstract)
Allow background thread process of a variable TThreadMethod callback

procedure RunAndWait(Method: TThreadMethod); reintroduce;
Run once the supplied TThreadMethod callback
- use this method, and not the inherited RunAndWait()

TSynBackgroundThreadProcedure = class(TSynBackgroundThreadMethodAbstract)
Allow background thread process of a procedure callback

constructor Create(aOnProcess: TOnProcessSynBackgroundThreadProc; aOnIdle: TOnIdleSynBackgroundThread; const aThreadName: RawUTF8); reintroduce;
Initialize the thread
- if aOnIdle is not set (i.e. equals nil), it will simply wait for the background process to finish until RunAndWait() will return

  Provide a procedure handler to be execute in the background thread
  - triggered by RunAndWait() method - which will wait until finished
  - the OpaqueParam as specified to RunAndWait() will be supplied here

ESynParallelProcess = class(ESynException)

  An exception which would be raised by TSynParallelProcess

TSynParallelProcessThread = class(TSynBackgroundThreadMethodAbstract)

  Thread executing process for TSynParallelProcess

TSynParallelProcess = class(TSynPersistentLock)

  Allow parallel execution of an index-based process in a thread pool
  - will create its own thread pool, then execute any method by splitting the work into each thread

constructor Create(ThreadPoolCount: integer; const ThreadName: RawUTF8;
  OnBeforeExecute: TNotifyThreadEvent=nil; OnAfterExecute: TNotifyThreadEvent=nil;
  MaxThreadPoolCount: integer = 32); reintroduce; virtual;

  initialize the thread pool
  - you could define some callbacks to nest the thread execution, e.g. assigned to
    TSQLRestServer.BeginCurrentThread/EndCurrentThread
  - up to MaxThreadPoolCount=32 threads could be setup (you may allow a bigger value, but
    interest of this thread pool is to have its process saturating each CPU core)
  - if ThreadPoolCount is 0, no thread would be created, and process would take place in the
current thread

destructor Destroy; override;

  Finalize the thread pool

procedure ParallelRunAndWait(const Method: TSynParallelProcessMethod; MethodCount:
  integer; const OnMainThreadIdle: TNotifyEvent = nil); reintroduce;

  Run a method in parallel, and wait for the execution to finish
  - will split Method[0..MethodCount-1] execution over the threads
  - in case of any exception during process, an ESynParallelProcess exception would be raised by
    this method
  - if OnMainThreadIdle is set, the current thread (which is expected to be e.g. the main UI thread)
    won't process anything, but call this event during waiting for the background threads

property ParallelRunCount: integer read fParallelRunCount;

  How many threads have been activated

property ThreadName: RawUTF8 read fThreadName;

  Some text identifier, used to distinguish each owned thread

property ThreadPoolCount: integer read fThreadPoolCount;

  How many threads are currently in this instance thread pool

TSynBackgroundThreadProcess = class(TSynBackgroundThreadAbstract)

  TThread able to run a method at a given periodic pace
constructor Create(const aThreadName: RawUTF8; aOnProcess: TNotifyThreadEvent=nil; aOnBeforeExecute: TNotifyThreadEvent=nil; aOnAfterExecute: TNotifyThreadEvent=nil; aStats: TSynMonitorClass=nil; CreateSuspended: boolean=false); reintroduce; virtual;

Initialize the thread for a periodic task processing
- aOnProcess would be called when ProcessEvent.SetEvent is called or aOnProcessMS milliseconds period was elapsed since last process
- if aOnProcessMS is 0, will wait until ProcessEvent.SetEvent is called
- you could define some callbacks to nest the thread execution, e.g. assigned to TSQLRestServer.BeginCurrentThread/EndCurrentThread

destructor Destroy; override;

Finalize the thread

property OnException: TNotifyEvent read fOnException write fOnException;

Event callback executed when OnProcess did raise an exception
- supplied Sender parameter is the raised Exception instance

property OnProcess: TOnSynBackgroundThreadProcess read fOnProcess;

Access to the implementation event of the periodic task

property OnProcessMS: cardinal read fOnProcessMS write fOnProcessMS;

Access to the delay, in milliseconds, of the periodic task processing

property Stats: TSynMonitor read fStats;

Processing statistics
- may be nil if aStats was nil in the class constructor

TSynBackgroundTimerTask = record

Used by TSynBackgroundTimer internal registration list

TSynBackgroundTimer = class(TSynBackgroundThreadProcess)

TThread able to run one or several tasks at a periodic pace in a background thread
- as used e.g. by TSQLRest.TimerEnable/TimerDisable methods, via the inherited TSQLRestBackgroundTimer
- each process can have its own FIFO of text messages
- if you expect to update some GUI, you should rather use a TTimer component (with a period of e.g. 200ms), since TSynBackgroundTimer will use its own separated thread

constructor Create(const aThreadName: RawUTF8; aOnBeforeExecute: TNotifyThreadEvent=nil; aOnAfterExecute: TNotifyThreadEvent=nil; aStats: TSynMonitorClass=nil; CreateSuspended: boolean=false); reintroduce; virtual;

Initialize the thread for a periodic task processing
- you could define some callbacks to nest the thread execution, e.g. assigned to TSQLRestServer.BeginCurrentThread/EndCurrentThread, as made by TSQLRestBackgroundTimer.Create

destructor Destroy; override;

Finalize the thread
function DeQueue(aOnProcess: TOnSynBackgroundTimerProcess; const aMsg: RawUTF8): boolean;

Remove a message from the processing list
- supplied message will be searched in the internal FIFO list associated with aOnProcess, then
- aOnProcess should have been registered by a previous call to Enable() method
- returns true on success, false if the supplied message was not registered

function Disable(aOnProcess: TOnSynBackgroundTimerProcess): boolean;

Undefined a task running on a periodic number of seconds
- aOnProcess should have been registered by a previous call to Enable() method
- returns true on success, false if the supplied task was not registered
- for background process on a mORMot service, consider using TSQLRestServer
  TimerEnable/TimerDisable methods, and their TSynBackgroundTimer thread

function EnQueue(aOnProcess: TOnSynBackgroundTimerProcess; const aMsgFmt: RawUTF8;
  const Args: array of const; aExecuteNow: boolean = false): boolean; overload;

Add a message to be processed during the next execution of a task
- supplied message will be added to the internal FIFO list associated with aOnProcess, then
- supplied to as aMsg parameter for each call
- if aExecuteNow is true, won't wait for the next aOnProcessSecs occurrence
- aOnProcess should have been registered by a previous call to Enable() method
- returns true on success, false if the supplied task was not registered

function EnQueue(aOnProcess: TOnSynBackgroundTimerProcess; const aMsg: RawUTF8;
  aExecuteNow: boolean = false): boolean; overload;

Add a message to be processed during the next execution of a task
- supplied message will be added to the internal FIFO list associated with aOnProcess, then
- supplied to as aMsg parameter for each call
- if aExecuteNow is true, won't wait for the next aOnProcessSecs occurrence
- aOnProcess should have been registered by a previous call to Enable() method
- returns true on success, false if the supplied task was not registered

function ExecuteNow(aOnProcess: TOnSynBackgroundTimerProcess): boolean;

Execute a task without waiting for the next aOnProcessSecs occurrence
- aOnProcess should have been registered by a previous call to Enable() method
- returns true on success, false if the supplied task was not registered

function Processing: boolean;

Returns true if there is currently one task processed

procedure Enable(aOnProcess: TOnSynBackgroundTimerProcess; aOnProcessSecs: cardinal);

Define a process method for a task running on a periodic number of seconds
- for background process on a mORMot service, consider using TSQLRest
  TimerEnable/TimerDisable methods, and its associated BackgroundTimer thread

procedure WaitUntilNotProcessing(timeoutsecs: integer = 10);

Wait until no background task is processed

property Task: TSynBackgroundTimerTaskDynArray read fTask;

Low-level access to the internal task list
property TaskLock: TSynLocker read fTaskLock;

Low-level access to the internal task mutex

TBlockingProcess = class(TEvent)

A semaphore used to wait for some process to be finished
- used e.g. by TBlockingCallback in mORMot.pas
- once created, process would block via a WaitFor call, which would be released when NotifyFinished is called by the process background thread

constructor Create(aTimeOutMs: integer); reintroduce; overload; virtual;

Initialize the semaphore instance
- specify a time out milliseconds period after which blocking execution should be handled as failure (if 0 is set, default 3000 would be used)
- an associated mutex would be created and owned by this instance

constructor Create(aTimeOutMs: integer; aSafe: PSynLocker); reintroduce; overload; virtual;

Override to reset associated params initialize the semaphore instance
- specify a time out milliseconds period after which blocking execution should be handled as failure (if 0 is set, default 3000 would be used)
- an associated mutex shall be supplied

destructor Destroy; override;

Finalize the instance

function NotifyFinished(alreadyLocked: boolean=false): boolean; virtual;

Should be called by the background process when it is finished
- the caller would then let its WaitFor method return
- returns TRUE on success (i.e. status was not evRaised or evTimeout)
- if the instance is already locked (e.g. when retrieved from TBlockingProcessPool.FromCallLocked), you may set alreadyLocked=TRUE

function Reset: boolean; virtual;

Just a wrapper to reset the internal Event state to evNone
- may be used to re-use the same TBlockingProcess instance, after a successfull WaitFor/NotifyFinished process
- returns TRUE on success (i.e. status was not evWaiting), setting the current state to evNone, and the Call property to 0
- if there is a WaitFor currently in progress, returns FALSE

function WaitFor: TBlockingEvent; reintroduce; overload; virtual;

Called to wait for NotifyFinished() to be called, or trigger timeout
- returns the final state of the process, i.e. evRaised or evTimeOut

function WaitFor(TimeOutMS: integer): TBlockingEvent; reintroduce; overload;

Called to wait for NotifyFinished() to be called, or trigger timeout
- returns the final state of the process, i.e. evRaised or evTimeOut

procedure Lock;

Just a wrapper around fSafe^.Lock
procedure Unlock;
  Just a wrapper around fSafe^.Unlock

property Event: TBlockingEvent read fEvent;
  The current state of process
  - use Reset method to re-use this instance after a WaitFor process

property TimeoutMs: integer read fTimeoutMS;
  The timeout period, in ms, as defined at constructor level

TBlockingProcessPoolItem = class(TBlockingProcess)
  A semaphore used in the TBlockingProcessPool
  - such semaphore have a Call field to identify each execution

property Call: TBlockingProcessPoolCall read fCall;
  An unique identifier, when owned by a TBlockingProcessPool
  - Reset would restore this field to its 0 default value

TBlockingProcessPool = class(TSynPersistent)
  Manage a pool of TBlockingProcessPoolItem instances
  - each call will be identified via a TBlockingProcessPoolCall unique value
  - it would also allow to re-use TEvent system resources

constructor Create(aClass: TBlockingProcessPoolItemClass=nil); reintroduce;
  Set TBlockingProcessPoolItem.Call initialize the pool, for a given implementation class

destructor Destroy; override;
  Finalize the pool
  - would also force all pending WaitFor to trigger a evTimeOut

function FromCall(call: TBlockingProcessPoolCall; locked: boolean=false): TBlockingProcessPoolItem; virtual;
  Retrieve a TBlockingProcess from its call identifier
  - may be used e.g. from the callback of the asynchronous process to set some additional
    parameters to the inherited TBlockingProcess, then call NotifyFinished to release the caller
    WaitFor
  - if leavelocked is TRUE, the returned instance would be locked: caller should execute
    result.Unlock or NotifyFinished(true) after use

function NewProcess(aTimeOutMs: integer): TBlockingProcessPoolItem; virtual;
  Book a TBlockingProcess from the internal pool
  - returns nil on error (e.g. the instance is destroying)
  - or returns the blocking process instance corresponding to this call; its Call property would
    identify the call for the asynchronous callback, then after WaitFor, the Reset method should be
    run to release the mutex for the pool
**TSystemUseData = packed record**

*Store CPU and RAM usage for a given process*
- as used by TSystemUse class

- **Kernel: single;**
  *Percent of current Kernel-space CPU usage for this process*

- **Timestamp: TDateTime;**
  *When the data has been sampled*

- **User: single;**
  *Percent of current User-space CPU usage for this process*

- **VirtualKB: cardinal;**
  *How many KB of virtual memory are used by this process*

- **WorkKB: cardinal;**
  *How many KB of working memory are used by this process*

**TProcessInfo = object(TObject)**

*Low-level structure used to compute process memory and CPU usage*

- **function Init: boolean;**
  *Initialize the system/process resource tracking*

- **function PerProcess(PID: cardinal; Now: PDateTime; out Data: TSystemUseData; var PrevKernel, PrevUser: Int64): boolean;**
  *Retrieve CPU and RAM usage for a given process*

- **function PerSystem(out Idle,Kernel,User: currency): boolean;**
  *Percent of current Idle/Kernel/User CPU usage for all processes*

- **function Start: boolean;**
  *To be called before PerSystem() or PerProcess() iteration*

**TSystemUseProcess = record**

*Internal storage of CPU and RAM usage for one process*

**TSystemUse = class(TSynPersistentLock)**

*Monitor CPU and RAM usage of one or several processes*
- you should execute BackgroundExecute on a regular pace (e.g. every second) to gather low-level CPU and RAM information for the given set of processes
- is able to keep an history of latest sample values
- use Current class function to access a process-wide instance
constructor Create(const aProcessID: array of integer; aHistoryDepth: integer=60); reintroduce; overload; virtual;
  Track the CPU and RAM usage of the supplied set of Process ID
  - any aProcessID[],=0 will be replaced by the current process ID
  - you can specify the number of sample values for the History() method
  - you should then execute the BackgroundExecute method of this instance in a VCL timer or
    from a TSynBackgroundTimer.Enable() registration

constructor Create(aHistoryDepth: integer=60); reintroduce; overload; virtual;
  Track the CPU and RAM usage of the current process
  - you can specify the number of sample values for the History() method
  - you should then execute the BackgroundExecute method of this instance in a VCL timer or
    from a TSynBackgroundTimer.Enable() registration

class function Current(aCreateIfNone: boolean=true): TSystemUse;
  Access to a global instance, corresponding to the current process
  - its HistoryDepth will be of 60 items

function Data(aProcessID: integer=0): TSystemUseData; overload;
  Returns the detailed CPU and RAM usage percent of the supplied process
  - aProcessID=0 will return information from the current process
  - returns Timestamp=0 if the Process ID was not registered via Create/Subcribe

function Data(out aData: TSystemUseData; aProcessID: integer=0): boolean; overload;
  Returns the detailed CPU and RAM usage percent of the supplied process
  - aProcessID=0 will return information from the current process
  - returns -1 if the Process ID was not registered via Create/Subcribe

function History(aProcessID: integer=0; aDepth: integer=0): TSingleDynArray;
  Returns total (Kernel+User) CPU usage percent history of the supplied process
  - aProcessID=0 will return information from the current process
  - returns nil if the Process ID was not registered via Create/Subcribe
  - returns the sample values as an array, starting from the last to the oldest
  - you can customize the maximum depth, with aDepth < HistoryDepth

function HistoryData(aProcessID: integer=0; aDepth: integer=0): TSystemUseDataDynArray; overload;
  Returns detailed CPU and RAM usage history of the supplied process
  - aProcessID=0 will return information from the current process
  - returns nil if the Process ID was not registered via Create/Subcribe
  - returns the sample values as an array, starting from the last to the oldest
  - you can customize the maximum depth, with aDepth < HistoryDepth

function HistoryText(aProcessID: integer=0; aDepth: integer=0; aDestMemoryMB: PRawUTF8=nil): RawUTF8;
  Returns total (Kernel+User) CPU usage percent history of the supplied process, as a string of two
digits values
  - aProcessID=0 will return information from the current process
  - returns "" if the Process ID was not registered via Create/Subcribe
  - you can customize the maximum depth, with aDepth < HistoryDepth
  - the memory history (in MB) can be optionally returned in aDestMemoryMB
function HistoryVariant(aProcessID: integer=0; aDepth: integer=0): variant;
Returns total (Kernel+User) CPU usage percent history of the supplied process
- aProcessID=0 will return information from the current process
- returns null if the Process ID was not registered via Create/Subscribe
- returns the sample values as a TDocVariant array, starting from the last to the oldest, with two digits precision (as currency values)
- you can customize the maximum depth, with aDepth < HistoryDepth

function KB(aProcessID: integer=0): cardinal; overload;
Returns the total (Work+Paged) RAM use of the supplied process, in KB
- aProcessID=0 will return information from the current process
- returns 0 if the Process ID was not registered via Create/Subscribe

function Percent(aProcessID: integer=0): single; overload;
Returns the total (Kernel+User) CPU usage percent of the supplied process
- aProcessID=0 will return information from the current process
- returns -1 if the Process ID was not registered via Create/Subscribe

function PercentKernel(aProcessID: integer=0): single; overload;
Returns the Kernel-space CPU usage percent of the supplied process
- aProcessID=0 will return information from the current process
- returns -1 if the Process ID was not registered via Create/Subscribe

function PercentSystem(out Idle,Kernel,User: currency): boolean;
Percent of current Idle/Kernel/User CPU usage for all processes

function PercentUser(aProcessID: integer=0): single; overload;
Returns the User-space CPU usage percent of the supplied process
- aProcessID=0 will return information from the current process
- returns -1 if the Process ID was not registered via Create/Subscribe

function Unsubscribe(aProcessID: integer): boolean;
Remove a Process ID from the internal tracking list

procedure BackgroundExecute(Sender: TSynBackgroundTimer; Event: TWaitResult; const Msg: RawUTF8);
A TSynBackgroundThreadProcess compatible event
- matches TOnSynBackgroundTimerProcess callback signature
- to be supplied e.g. to a TSynBackgroundTimer.Enable method so that it will run every few seconds and retrieve the CPU and RAM use

procedure OnTimerExecute(Sender: TObject);
A VCL’s TTimer.OnTimer compatible event
- to be run every few seconds and retrieve the CPU and RAM use:
  tmrSystemUse.Interval := 10000; // every 10 seconds

procedure Subscribe(aProcessID: integer);
Add a Process ID to the internal tracking list

property HistoryDepth: integer read fHistoryDepth;
How many items are stored internally, and returned by the History() method
property OnMeasured: TOnSystemUseMeasured read fOnMeasured write fOnMeasured;
   Executed when TSystemUse.BackgroundExecute finished its measurement

property Timer: TSynBackgroundTimer read fTimer write fTimer;
   Low-level access to the associated timer running BackgroundExecute
   - equals nil if has been associated to no timer

   If any unexisting (e.g. closed/killed) process should be unregistered
   - e.g. if OpenProcess() API call fails

TDiskPartition = packed record
   Stores information about a disk partition
   mounted: TFileName;
      Where this partition has been mounted
      - e.g. 'C:' or '/home'
      - you can use GetDiskInfo(mounted) to retrieve current space information
   name: RawUTF8;
      The name of this partition
      - is the Volume name under Windows, or the Device name under POSIX
   size: QWord;
      Total size (in bytes) of this partition

TSynMonitorMemory = class(TSynPersistent)
   Value object able to gather information about the current system memory
   constructor Create(aTextNoSpace: boolean); reintroduce;
      Initialize the class, and its nested TSynMonitorOneSize instances
   destructor Destroy; override;
      Finalize the class, and its nested TSynMonitorOneSize instances
   class function FreeAsText(nospace: boolean=false): ShortString;
      Some text corresponding to current 'free/total' memory information
      - returns e.g. '10.3 GB / 15.6 GB'
   class function PhysicalAsText(nospace: boolean=false): TShort16;
      How many physical memory is currently installed, as text (e.g. '32 GB');
   class function ToJSON: RawUTF8;
      Returns a JSON object with the current system memory information
      - numbers would be given in KB (Bytes shl 10)
   class function ToVariant: variant;
      Fill a TDocVariant with the current system memory information
      - numbers would be given in KB (Bytes shl 10)
property AllocatedReserved: TSynMonitorOneSize read GetAllocatedReserved;
  Total of allocated memory reserved by the program

property AllocatedUsed: TSynMonitorOneSize read GetAllocatedUsed;
  Total of allocated memory used by the program

property MemoryLoadPercent: integer read GetMemoryLoadPercent;
  Percent of memory in use for the system

property PagingFileFree: TSynMonitorOneSize read GetPagingFileFree;
  Free of paging file for the system

property PagingFileTotal: TSynMonitorOneSize read GetPagingFileTotal;
  Total of paging file for the system

property PhysicalMemoryFree: TSynMonitorOneSize read GetPhysicalMemoryFree;
  Free of physical memory for the system

property PhysicalMemoryTotal: TSynMonitorOneSize read GetPhysicalMemoryTotal;
  Total of physical memory for the system

property VirtualMemoryFree: TSynMonitorOneSize read GetVirtualMemoryFree;
  Free of virtual memory for the system
- property not defined under Linux, since not applying to this OS

property VirtualMemoryTotal: TSynMonitorOneSize read GetVirtualMemoryTotal;
  Total of virtual memory for the system
- property not defined under Linux, since not applying to this OS

TSynMonitorDisk = class(TSynPersistent)
Value object able to gather information about a system drive

constructor Create; override;
  Initialize the class, and its nested TSynMonitorOneSize instances

destructor Destroy; override;
  Finalize the class, and its nested TSynMonitorOneSize instances

class function FreeAsText: RawUTF8;
  Some text corresponding to current 'free/total' disk information
  - could return e.g. 'D: 64.4 GB / 213.4 GB'

property AvailableSize: TSynMonitorOneSize read GetAvailable;
  Space currently available on this disk for the current user
  - may be less then FreeSize, if user quotas are specified (only taken into account under Windows)

property FreeSize: TSynMonitorOneSize read GetFree;
  Free space currently available on this disk

property Name: TFileName read GetName;
  The disk name
property TotalSize: TSynMonitorOneSize read GetTotal;
  Total space

property VolumeName: TFileName read fVolumeName write fVolumeName;
  The volume name (only available on Windows)

TMemoryInfo = record
  Hold low-level information about current memory usage
  - as filled by GetMemoryInfo()

TTimeZoneInfo = record
  Used to store Time Zone bias in TSynTimeZone
  - map how low-level information is stored in the Windows Registry

TTimeZoneData = object(TObject)
  Used to store Time Zone information for a single area in TSynTimeZone
  - Delphi "object" is buggy on stack -> also defined as record with methods

TSynTimeZone = class(TObject)
  Handle cross-platform time conversions, following Microsoft time zones
  - is able to retrieve accurate information from the Windows registry, or from a binary compressed file on other platforms (which should have been saved from a Windows system first)
  - each time zone will be identified by its TzId string, as defined by Microsoft for its Windows Operating system

constructor Create;
  Initialize the internal storage
  - but no data is available, until Load* methods are called

constructor CreateDefault(dummy: integer=0);
  Retrieve the time zones from Windows registry, or from a local file
  - under Linux, the file should be located with the executable, renamed with a .tz extension - may have been created via SaveToFile(''), or from a 'TSynTimeZone' bound resource "dummy"
  parameter exists only to disambiguate constructors for C++

destructor Destroy; override;
  Finalize the instance

class function Default: TSynTimeZone;
  Will retrieve the default shared TSynTimeZone instance
  - locally created via the CreateDefault constructor
  - this is the usual entry point for time zone process, calling e.g.
    aLocalTime := TSynTimeZone.Default.NowToLocal(aTimeZoneID);

function Displays: TStrings;
  Returns a TStringList of all Display text values
  - could be used to fill any VCL component to select the time zone
  - order in Displays[] array follows the Zone[].display information
function GetBiasForDateTime(const Value: TDateTime; const TzId: TTimeZoneID; out Bias: integer; out HaveDaylight: boolean): boolean;

Retrieve the time bias (in minutes) for a given date/time on a TzId

function GetDisplay(const TzId: TTimeZoneID): RawUTF8;

Retrieve the display text corresponding to a TzId
- returns '' if the supplied TzId is not recognized

function Ids: TStrings;

Returns a TStringList of all TzID values
- could be used to fill any VCL component to select the time zone
- order in Ids[] array follows the Zone[].id information

function LocalToUtc(const LocalDateTime: TDateTime; const TzID: TTimeZoneID): TDateTime;

Compute the UTC date/time for a given local TzId value
- by definition, a local time may correspond to two UTC times, during the time biais period, so
the returned value is informative only, and any stored value should be following UTC

function NowToLocal(const TzId: TTimeZoneID): TDateTime;

Compute the current date/time corrected for a given TzId

function SaveToBuffer: RawByteString;

Write then time zone information into a compressed memory buffer

function UtcToLocal(const UtcDateTime: TDateTime; const TzId: TTimeZoneID): TDateTime;

Compute the UTC date/time corrected for a given TzId

procedure LoadFromBuffer(const Buffer: RawByteString);

Read time zone information from a compressed memory buffer

procedure LoadFromFile(const FileName: TFileName='');

Read time zone information from a compressed file
- if no file name is supplied, a ExecutableName.tz file would be used

procedure LoadFromRegistry;

Read time zone information from the Windows registry

procedure LoadFromResource(Instance: THandle=0);

Read time zone information from a 'TSynTimeZone' resource
- the resource should contain the SaveToBuffer compressed binary content
- is no resource matching the TSynTimeZone class name and ResType=10 do exist, nothing would be
loaded
- the resource could be created as such, from a Windows system:
  TSynTimeZonE.Default.SaveToFile('TSynTimeZone.data');
  then compile the resource as expected, with a brcc32 .rc entry:
  TSynTimeZone 10 "TSynTimeZone.data"
- you can specify a library (dll) resource instance handle, if needed

procedure SaveToFile(const FileName: TFileName);

Write then time zone information into a compressed file
- if no file name is supplied, a ExecutableName.tz file would be created
```plaintext

**property** Zone: TTimeZoneDataDynArray read fZone;
Direct access to the low-level time zone information

**property** Zones: TDynArrayHashed read fZones;
Direct access to the wrapper over the time zone information array

ICommandLine = interface(IInterface)
An interface to process the command line switches over a console
- as implemented e.g. by TCommandLine class
- can implement any process, optionally with console interactivity

**function** AsArray: TRawUTF8DynArray;
Returns all command line values as an array of UTF-8 text
- i.e. won't interpret the various switches in the input parameters
- as created e.g. by TCommandLine.CreateAsArray constructor

**function** AsDate(const Switch: RawUTF8; Default: TDateTime; const Prompt: string): TDateTime;
Returns a command line switch ISO-8601 value as date value
- here dates are expected to be encoded with ISO-8601, i.e. YYYY-MM-DD
- you can specify a prompt text, when asking for any missing switch

**function** AsEnum(const Switch: RawUTF8; Default: RawUTF8; TypeInfo: pointer; const Prompt: string): integer;
Returns a command line switch value as enumeration ordinal
- RTTI will be used to check for the enumeration text, or plain integer value will be returned as ordinal value
- you can specify a prompt text, when asking for any missing switch

**function** AsInt(const Switch: RawUTF8; Default: Int64; const Prompt: string): Int64;
Returns a command line switch value as integer
- you can specify a prompt text, when asking for any missing switch

**function** AsJSON(Format: TTextWriterJSONFormat): RawUTF8;
Serialize all recognized switches as UTF-8 JSON text

**function**AsString(const Switch: RawUTF8; const Default, Prompt: string): string;
Returns a command line switch value as VCL string text
- you can specify a prompt text, when asking for any missing switch

**function** AsUTF8(const Switch, Default: RawUTF8; const Prompt: string): RawUTF8;
Returns a command line switch value as UTF-8 text
- you can specify a prompt text, when asking for any missing switch

**function** NoPrompt: boolean;
Equals TRUE if the -noprompt switch has been supplied
- may be used to force pure execution without console interaction, e.g. when run from another process
```
procedure Text(const Fmt: RawUTF8; const Args: array of const; Color: TConsoleColor=ccLightGray);

Write some console text, with an optional color
- will output the text even if NoPrompt is TRUE

procedure TextColor(Color: TConsoleColor);

Change the console text color
- do nothing if NoPrompt is TRUE

TCommandLine = class(TInterfacedObjectWithCustomCreate)

A class to process the command line switches, with console interactivity
- is able to redirect all Text() output to an internal UTF-8 storage, in addition or instead of the console (to be used e.g. from a GUI)
- implements ICommandLine interface

constructor Create(const switches: variant; aNoConsole: boolean=true); reintroduce; overload;

Initialize the internal storage with some ready-to-use switches
- will also set the NoPrompt option, and set the supplied NoConsole value
- may be used e.g. from a graphical interface instead of console mode

constructor Create(const NameValuePairs: array of const; aNoConsole: boolean=true); reintroduce; overload;

Initialize the internal storage with some ready-to-use name/value pairs
- will also set the NoPrompt option, and set the supplied NoConsole value
- may be used e.g. from a graphical interface instead of console mode

constructor Create; overload; override;

Initialize the internal storage from the command line
- will parse "-switch1 value1 -switch2 value2" layout
- stand-alone "-switch1 -switch2 value2" will a create switch1=true value

constructor CreateAsArray(firstParam: integer);

Initialize the internal storage from the command line
- will set paramstr(firstParam)..paramstr(paramcount) in fValues as array
- may be used e.g. for "val1 val2 val3" command line layout

function AsArray: TRawUTF8DynArray;

Returns all command line values as an array of UTF-8 text
- i.e. won't interpret the various switches in the input parameters
- as created e.g. by TCommandLine.CreateAsArray constructor

function AsDate(const Switch: RawUTF8; Default: TDateTime; const Prompt: string): TDateTime;

Returns a command line switch ISO-8601 value as date value
- here dates are expected to be encoded with ISO-8601, i.e. YYYY-MM-DD
- you can specify a prompt text, when asking for any missing switch
function AsEnum(const Switch, Default: RawUTF8; TypeInfo: pointer; const Prompt: string): integer;
  Returns a command line switch value as enumeration ordinal
  - RTTI will be used to check for the enumeration text, or plain integer value will be returned as ordinal value
  - you can specify a prompt text, when asking for any missing switch

function AsInt(const Switch: RawUTF8; Default: Int64; const Prompt: string): Int64;
  Returns a command line switch value as integer
  - you can specify a prompt text, when asking for any missing switch

function AsJSON(Format: TTextWriterJSONFormat): RawUTF8;
  Serialize all recognized switches as UTF-8 JSON text

function AsString(const Switch: RawUTF8; const Default, Prompt: string): string;
  Returns a command line switch value as VCL string text
  - you can specify a prompt text, when asking for any missing switch

function AsUTF8(const Switch, Default: RawUTF8; const Prompt: string): RawUTF8;
  Returns a command line switch value as UTF-8 text
  - you can specify a prompt text, when asking for any missing switch

function ConsoleText(const LineFeed: RawUTF8=sLineBreak): RawUTF8;
  Returns the UTF-8 text as inserted by Text() calls
  - line feeds will be included to the ConsoleLines[] values

function NoPrompt: boolean;
  Equals TRUE if the -noprompt switch has been supplied
  - may be used to force pure execution without console interaction, e.g. when run from another process

procedure Text(const Fmt: RawUTF8; const Args: array of const; Color: TConsoleColor=ccLightGray);
  Write some console text, with an optional color
  - will output the text even if NoPrompt=TRUE, but not if NoConsole=TRUE
  - will append the text to the internal storage, available from ConsoleText

procedure TextColor(Color: TConsoleColor);
  Change the console text color
  - do nothing if NoPrompt is TRUE

property ConsoleLines: TRawUTF8DynArray read fLines;
  Low-level access to the internal UTF-8 console lines storage

property NoConsole: boolean read fNoConsole write SetNoConsole;
  If Text() should be redirected to ConsoleText internal storage
  - and don't write anything to the console
  - should be associated with NoProperty = TRUE property

property Values: TDocVariantData read fValues;
  Low-level access to the internal switches storage

ETableDataException = class(ESynException)
  Exception raised by all TSynTable related code
TSynTableStatementSelect = record
  One recognized SELECT expression for TSynTableStatement
  Alias: RawUTF8;
    The optional column alias, e.g. 'MaxID' for 'max(id) as MaxID'
  Field: integer;
    The column SELECTed for the SQL statement, in the expected order
    - contains 0 for ID/RowID, or the RTTI field index + 1
  FunctionKnown: (funcNone, funcCountStar, funcDistinct, funcMax);
    If the function needs a special process
    - e.g. funcCountStar for the special Count( * ) expression or funcDistinct, funcMax for
distinct(...)/max(...) aggregation
  FunctionName: RawUTF8;
    The optional function applied to the SELECTed column
    - e.g. Max(RowID) would store 'Max' and SelectField[0]=0
    - but Count( * ) would store 'Count' and SelectField[0]=0, and set FunctionIsCountStar = TRUE
  SubField: RawUTF8;
    MongoDB-like sub field e.g. 'mainfield.subfield1.subfield2'
    - still identifying 'mainfield' in Field index, and setting SubField='.subfield1.subfield2'
  ToBeAdded: integer;
    An optional integer to be added
    - recognized from .. +123 .. -123 patterns in the select

TSynTableStatementWhere = record
  One recognized WHERE expression for TSynTableStatement
  Field: integer;
    The index of the field used for the WHERE expression
    - WhereField=0 for ID, 1 for field # 0, 2 for field #1, and so on... (i.e. WhereField = RTTI field index +1)
  FunctionName: RawUTF8;
    The SQL function name associated to a Field and Value
    - e.g. 'INTEGERDYNARRAYCONTAINS' and Field=0 for IntegerDynArrayContains(RowID,10) and
      ValueInteger=10
    - Value does not contain anything
  JoinedOR: boolean;
    Expressions are evaluated as AND unless this field is set to TRUE
  NotClause: boolean;
    If this expression is preceded by a NOT modifier
  Operator: TSynTableStatementOperator;
    The operator of the WHERE expression
ParenthesisAfter: RawUTF8;

Any ')' after the actual expression

ParenthesisBefore: RawUTF8;

Any '(' before the actual expression

SubField: RawUTF8;

MongoDB-like sub field e.g. 'mainfield.subfield1.subfield2'
- still identifying 'mainfield' in Field index, and setting SubField='.subfield1.subfield2'

Value: RawUTF8;

The value used for the WHERE expression

ValueInteger: integer;

An integer representation of WhereValue (used for ID check e.g.)

ValueSBF: TSBFString;

Used to fast compare with SBF binary compact formatted data

ValueSQL: PUTF8Char;

The raw value SQL buffer used for the WHERE expression

ValueSQLLen: integer;

The raw value SQL buffer length used for the WHERE expression

ValueVariant: variant;

The value used for the WHERE expression, encoded as Variant
- may be a TDocVariant for the IN operator

TSynTableStatement = class(TObject)

Used to parse a SELECT SQL statement, following the SQLite3 syntax
- handle basic REST commands, i.e. a SELECT over a single table (no JOIN) with its WHERE clause, and result column aliases
- handle also aggregate functions like "SELECT Count( * ) FROM TableName"
- will also parse any LIMIT, OFFSET, ORDER BY, GROUP BY statement clause

constructor Create(const SQL: RawUTF8; GetFieldIndex: TSynTableFieldIndex;
SimpleFieldsBits: TSQLFieldBits=[0..MAX_SQLFIELDS-1]; FieldProp:
TSynTableFieldProperties=nil);

Parse the given SELECT SQL statement and retrieve the corresponding parameters into this class read-only properties
- the supplied GetFieldIndex() method is used to populate the SelectedFields and Where[].Field properties
- SimpleFieldsBits is used for '*' field names
- SQLStatement is left '' if the SQL statement is not correct
- if SQLStatement is set, the caller must check for TableName to match the expected value, then use the Where[] to retrieve the content
- if FieldProp is set, then the Where[].ValueSBF property is initialized with the SBF equivalence of the Where[].Value
procedure SelectFieldBits(var Fields: TSQLFieldBits; var withID: boolean; SubFields: PRawUTF8Array=nil);

  Compute the SELECT column bits from the SelectFields array
  - optionally set Select[].SubField into SubFields[Select[].Field] (e.g. to include specific fields from
    MongoDB embedded document)

property GroupByField: TSQLFieldIndexDynArray read fGroupByField;
  Recognize an GROUP BY clause with one or several fields
  - here 0 = ID, otherwise RTTI field index +1

property HasSelectSubFields: boolean read fHasSelectSubFields;
  If any Select[].SubField was actually set

property Limit: integer read fLimit;
  The number specified by the optional LIMIT ... clause
  - set to 0 by default (meaning no LIMIT clause)

property Offset: integer read fOffset;
  The number specified by the optional OFFSET ... clause
  - set to 0 by default (meaning no OFFSET clause)

property OrderByDesc: boolean read fOrderByDesc;
  False for default ASC order, true for DESC attribute

property OrderByField: TSQLFieldIndexDynArray read fOrderByField;
  Recognize an ORDER BY clause with one or several fields
  - here 0 = ID, otherwise RTTI field index +1

property Select: TSynTableStatementSelectDynArray read fSelect;
  The column SELECTed for the SQL statement, in the expected order

property SelectFunctionCount: integer read fSelectFunctionCount;
  If the SELECTed expression of this SQL statement have any function defined

property SQLStatement: RawUTF8 read fSQLStatement;
  The SELECT SQL statement parsed
  - equals "" if the parsing failed

property TableName: RawUTF8 read fTableName;
  The retrieved table name

property Where: TSynTableStatementWhereDynArray read fWhere;
  The WHERE clause of this SQL statement

property WhereHasParenthesis: boolean read fWhereHasParenthesis;
  If the WHERE clause contains any ( ) parenthesis expression

property WhereHasSubFields: boolean read fWhereHasSubFields;
  If the WHERE clause contains any Where[].SubField

property Writer: TJSONWriter read fWriter write fWriter;
  Optional associated writer

TSortCompareTmp = record
  Internal value used by TSynTableFieldProperties.SortCompare() method to avoid stack allocation
TSynTableFieldProperties = class(TObject)

Store the type properties of a given field / database column

FieldNumber: integer;
Number of the field in the table (starting at 0)

FieldSize: integer;
The fixed-length size, or -1 for a varInt, -2 for a variable string

FieldType: TSynTableFieldType;
Kind of field (defines both value type and storage to be used)

Filters: TSynObjectList;
All TSynValidate instances registered per each field

Name: RawUTF8;
The field name

Offset: integer;
Contains the offset of this field, in case of fixed-length field
- normally, fixed-length fields are stored in the beginning of the record storage: in this case, a value >= 0 will point to the position of the field value of this field
- if the value is < 0, its absolute will be the field number to be counted after TSynTable.fFieldVariableOffset (-1 for first item)

Options: TSynTableFieldOptions;
Options of this field

OrderedIndex: TIntegerDynArray;
If allocated, contains the storage indexes of every item, in sorted order
- only available if tfoIndex is in Options
- the index is not the per-ID index, but the "physical" index, i.e. the index value used to retrieve data from low-level (and faster) method

OrderedIndexCount: integer;
Number of items in OrderedIndex[]
- is set to 0 when the content has been modified (mark force recreate)

OrderedIndexNotSorted: boolean;
If set to TRUE after an OrderedIndex[] refresh but with not sorting
- OrderedIndexSort(0,OrderedIndexCount-1) must be called before using the OrderedIndex[] array
- you should call_orderedIndexRefresh method to ensure it is sorted

OrderedIndexReverse: TIntegerDynArray;
If allocated, contains the reverse storage index of OrderedIndex
- i.e. OrderedIndexReverse[OrderedIndex[i]] := i;
- used to speed up the record update procedure with huge number of records

Validates: TSynObjectList;
All TSynValidate instances registered per each field
constructor CreateFrom(var RD: TFileBufferReader);
    Read entry from a specified file reader

destructor Destroy; override;
    Release associated memory and objects

function AddFilterOrValidate(aFilter: TSynFilterOrValidate): TSynFilterOrValidate;
    Register a custom filter or validation rule to the class for this field
    - this will be used by Filter() and Validate() methods
    - will return the specified associated TSynFilterOrValidate instance
    - a TSynValidateTableUniqueField is always added by TSynTable.AfterFieldModif if tfoUnique is set in Options

function GetBoolean(RecordBuffer: pointer): Boolean;
    Decode the value from a record buffer into an Boolean
    - will call Owner.GetData to retrieve then decode the field SBF content

function GetCurrency(RecordBuffer: pointer): Currency;
    Decode the value from a record buffer into an currency value
    - will call Owner.GetData to retrieve then decode the field SBF content

function GetDouble(RecordBuffer: pointer): Double;
    Decode the value from a record buffer into an floating-point value
    - will call Owner.GetData to retrieve then decode the field SBF content

function GetInt64(RecordBuffer: pointer): Int64;
    Decode the value from a record buffer into an Int64
    - will call Owner.GetData to retrieve then decode the field SBF content

function GetInteger(RecordBuffer: pointer): Integer;
    Decode the value from a record buffer into an integer
    - will call Owner.GetData to retrieve then decode the field SBF content

function GetJSON(FieldBuffer: pointer; W: TTextWriter): pointer;
    Decode the value from our SBF compact binary format into UTF-8 JSON
    - returns the next FieldBuffer value

function GetLength(FieldBuffer: pointer): Integer;
    Retrieve the binary length (in bytes) of some SBF compact binary format

function GetRawUTF8(RecordBuffer: pointer): RawUTF8;
    Decode the value from a record buffer into a RawUTF8 string
    - will call Owner.GetData to retrieve then decode the field SBF content

function GetValue(FieldBuffer: pointer): RawUTF8;
    Decode the value from our SBF compact binary format into UTF-8 text
    - this method does not check for FieldBuffer to be not nil -> caller should check this explicitly

function GetVariant(FieldBuffer: pointer): Variant; overload;
    Decode the value from our SBF compact binary format into a Variant
function OrderedIndexMatch(WhereSBFValue: pointer; var MatchIndex: TIntegerDynArray; var MatchIndexCount: integer; Limit: Integer=0): Boolean;

Retrieve one or more "physical" indexes matching a WHERE Statement
- is faster than O(1) Gettering(), because will use O(log(n)) binary search using the OrderedIndex[] array
- returns the resulting indexes as a a sorted list in MatchIndex/MatchIndexCount
- if the indexes are already present in the list, won't duplicate them
- WhereSBFValue must be a valid SBF formulated field buffer content
- the Limit parameter is similar to the SQL LIMIT clause: if greater than 0, an upper bound on the number of rows returned is placed (e.g. set Limit=1 to only retrieve the first match)
- GetData property must have been set with a method returning a pointer to the field data for a given index (this index is not the per-ID index, but the "physical" index, i.e. the index value used to retrieve data from low-level (and fast) GetData method)
- in this method, indexes are not the per-ID indexes, but the "physical" indexes, i.e. each index value used to retrieve data from low-level (and fast) GetData method

function OrderedIndexUpdate(aOldIndex, aNewIndex: integer; aOldRecordData, aNewRecordData: pointer): boolean;

Will update then sort the array of indexes used for the field index
- the OrderedIndex[] array is first refreshed according to the aOldIndex, aNewIndex parameters: aOldIndex=-1 for Add, aNewIndex=-1 for Delete, or both >= 0 for update
- call with both indexes = -1 will sort the existing OrderedIndex[] array
- GetData property must have been set with a method returning a pointer to the field data for a given index (this index is not the per-ID index, but the "physical" index, i.e. the index value used to retrieve data from low-level (and fast) GetData method)
- aOldRecordData and aNewRecordData can be specified in order to guess if the field data has really been modified (speed up the update a lot to only sort indexed fields if its content has been really modified)
- returns FALSE if any parameter is invalid

function SBF(const Value: Variant): TSBFString; overload;

Create some SBF compact binary format from a Variant value

function SBF(const Value: RawUTF8): TSBFString; overload;

Create some SBF compact binary format from a Delphi binary value
- expect a RawUTF8 string: will be converted to WinAnsiString before storage, for tftWinAnsi
- will return '' if the field type doesn't match a string

function SBF(Value: pointer; ValueLen: integer): TSBFString; overload;

Create some SBF compact binary format from a BLOB memory buffer
- will return '' if the field type doesn't match tftBlobInternal

function SBF(const Value: Int64): TSBFString; overload;

Create some SBF compact binary format from a Delphi binary value
- will encode any byte, word, integer, cardinal, Int64 value
- will return '' if the field type doesn't match an integer

function SBF(const Value: Boolean): TSBFString; overload;

Create some SBF compact binary format from a Delphi binary value
- will return '' if the field type doesn't match a boolean
function SBF(const Value: Integer): TSBFString; overload;
   // Create some SBF compact binary format from a Delphi binary value
   // - will encode any byte, word, integer, cardinal value
   // - will return " if the field type doesn't match an integer

function SBFCurr(const Value: Currency): TSBFString;
   // Create some SBF compact binary format from a Delphi binary value
   // - will return " if the field type doesn't match a currency
   // - we can't use SBF() method name because of Currency/Double ambiguity

function SBFFloat(const Value: Double): TSBFString;
   // Create some SBF compact binary format from a Delphi binary value
   // - will return " if the field type doesn't match a floating-point
   // - we can't use SBF() method name because of Currency/Double ambiguity

function SBFFromRawUTF8(const aValue: RawUTF8): TSBFString;
   // Convert any UTF-8 encoded value into our SBF compact binary format
   // - can be used e.g. from a WHERE clause, for fast comparison in
     TSynTableStatement.WhereValue content using OrderedIndex[]
   // - is the reverse of GetValue/GetRawUTF8 methods above

function SortCompare(P1, P2: PUTF8Char): PtrInt;
   // Low-level binary comparison used by IDSORT and TSynTable.IterateJSONValues
   // - P1 and P2 must point to the values encoded in our SBF compact binary format

function Validate(RecordBuffer: pointer; RecordIndex: integer): string;
   // Check the registered constraints
   // - returns " on success
   // - returns an error message e.g. if a tftUnique constraint failed
   // - RecordIndex=-1 in case of adding, or the physical index of the updated record

procedure GetVariant(FieldBuffer: pointer; var result: Variant); overload;
   // Decode the value from our SBF compact binary format into a Variant

procedure OrderedIndexRefresh;
   // Will force refresh the OrderedIndex[] array
   // - to be called e.g. if OrderedIndexNotSorted = TRUE, if you want to access to the OrderedIndex[]

procedure SaveTo(WR: TFileBufferWriter);
   // Save entry to a specified file writer

property SBFDefault: TSBFString read fDefaultFieldData;
   // Some default SBF compact binary format content

TSynTableData = object(TObject)
   // Used to store a TSynTable record using our SBF compact binary format
   // - this object can be created on the stack
   // - it is mapped into a variant TVarData, to be retrieved by the TSynTable.Data method - but direct
     allocation of a TSynTableData on the stack is faster (due to the Variant overhead)
   // - is defined either as an object either as a record, due to a bug in Delphi 2009/2010 compiler (at
     least): this structure is not initialized if defined as an object on the stack, but will be as a record :(
function GetFieldSBFValue(aField: TSynTableFieldProperties): TSBFString;
Get a field value for a specified field, into SBF-encoded data
- this method is faster than the other, because it won't look for the field name nor make any
variant conversion

function GetFieldValue(aField: TSynTableFieldProperties): Variant;
Get a field value for a specified field
- this method is faster than Field[], because it won't look for the field name

function ValidateSBFValue(RecordIndex: integer): string;
Check the registered constraints according to a record SBF buffer
- returns '' on success
- returns an error message e.g. if a tftUnique constraint failed
- RecordIndex=1 in case of adding, or the physical index of the updated record

procedure FilterSBFValue;
Filter the SBF buffer record content with all registered filters
- all field values are filtered in-place, following our SBF compact binary format encoding for this
record

procedure Init(aTable: TSynTable; aID: Integer=0); overload;
Initialize a record data content for a specified table
- a void content is set

procedure Init(aTable: TSynTable; aID: Integer; RecordBuffer: pointer;
RecordBufferLen: integer); overload;
Initialize a record data content for a specified table
- the specified SBF content is store inside this TSynTableData

procedure SetFieldSBFValue(aField: TSynTableFieldProperties; const
Value: TSBFString);
Set a field value for a specified field, from SBF-encoded data
- this method is faster than the other, because it won't look for the field name nor make any
variant conversion

procedure SetFieldValue(aField: TSynTableFieldProperties; const
Value: Variant);
Set a field value for a specified field
- this method is faster than Field[], because it won't look for the field name

property Field[const FieldName: RawUTF8]: Variant read GetField write SetField;
Set or retrieve a field value from a variant data

property ID: integer read VID write VID;
The associated record ID

property SBF: TSBFString read VValue;
The record content, SBF compact binary format encoded

property Table: TSynTable read VTable write VTable;
The associated TSynTable instance

TUpdateFieldEvent = record
An opaque structure used for TSynTable.UpdateFieldEvent method
AvailableFields: TSQLFieldBits;
The list of existing field in the previous data

Count: integer;
The number of record added

IDs: TIntegerDynArray;
The list of IDs added
- this list is already in increasing order, because GetIterating was called with the ioID order

NewIndexes: TIntegerDynArray;
Previous indexes: NewIndexes[oldIndex] := newIndex

Offsets64: TInt64DynArray;
The offset of every record added
- follows the IDs[] order

WR: TFileBufferWriter;
Where to write the updated data

TSynValidateTable = class(TSynValidate)
Will define a validation to be applied to a TSynTableFieldProperties field
- a typical usage is to validate a value to be unique in the table (implemented in the
  TSynValidateTableUniqueField class)
- the optional associated parameters are to be supplied JSON-encoded
- ProcessField and ProcessRecordIndex properties will be filled before Process method call by
  TSynTableFieldProperties.Validate()

property ProcessField: TSynTableFieldProperties read fProcessField write fProcessField;
The associated TSQRLRest instance
- this value is filled by TSynTableFieldProperties.Validate with its self value to be used for the
  validation
- it can be used in the overridden Process method

property ProcessRecordIndex: integer read fProcessRecordIndex write fProcessRecordIndex;
The associated record index (in case of update)
- is set to -1 in case of adding, or the physical index of the updated record
- this value is filled by TSynTableFieldProperties.Validate
- it can be used in the overridden Process method

TSynValidateTableUniqueField = class(TSynValidateTable)
Will define a validation for a TSynTableFieldProperties Unique field
- implement constraints check e.g. if tfoUnique is set in Options
- it will check that the field value is not void
- it will check that the field value is not a duplicate
function Process(aFieldIndex: integer; const Value: RawUTF8; var ErrorMsg: string): boolean; override;

Perform the unique field validation action to the specified value
- duplication value check will use the ProcessField and ProcessRecordIndex properties, which will be filled before call by TSynTableFieldProperties.Validate()
- aFieldIndex parameter is not used here, since we have already the ProcessField property set
- here the Value is expected to be UTF-8 text, as converted from our SBF compact binary format via e.g. TSynTableFieldProperties.GetValue / GetRawUTF8: this is mandatory to have the validation rule fit with other TSynValidateTable classes

TSynTable = class(TObject)

Store the description of a table with records, to implement a Database
- can be used with several storage engines, for instance TSynBigTableRecord
- each record can have up to 64 fields
- a mandatory ID field must be handled by the storage engine itself
- will handle the storage of records into our SBF compact binary format, in which fixed-length fields are stored leftmost side, then variable-length fields follow

constructor Create(const aTableName: RawUTF8);
Create a table definition instance

destructor Destroy; override;
Release used memory

function AddField(const aName: RawUTF8; aType: TSynTableFieldType; aOptions: TSynTableFieldOptions=[]): TSynTableFieldProperties;
Add a field description to the table
- warning: the class responsible of the storage itself must process the data already stored when a field is created, e.g. in TSynBigTableRecord.AddFieldUpdate method
- physical order does not necessary follow the AddField() call order: for better performance, it will try to store fixed-sized record first, multiple of 4 bytes first (access is faster if dat is 4 byte aligned), then variable-length after fixed-sized fields; in all case, a field indexed will be put first

function CreateJSONWriter(JSON: TStream; Expand, withID: boolean; const Fields: TSQLFieldIndexDynArray): TJSONWriter; overload;
Create a TJSONWriter, ready to be filled with GetJSONValues(W) below
- will initialize all TJSONWriter.ColNames[] values according to the specified Fields index list, and initialize the JSON content

function CreateJSONWriter(JSON: TStream; Expand, withID: boolean; const Fields: TSQLFieldBits): TJSONWriter; overload;
Create a TJSONWriter, ready to be filled with GetJSONValues(W) below
- will initialize all TJSONWriter.ColNames[] values according to the specified Fields bit set, and initialize the JSON content

function Data(aID: integer=0; RecordBuffer: pointer=nil; RecordBufferLen: Integer=0): Variant; overload;
Create a Variant able to access any field content via late binding
- i.e. you can use Var.Name to access the 'Name' field of record Var
- if you leave ID and RecordBuffer void, a void record is created
function DataLength(RecordBuffer: pointer): integer;
Return the total length of the given record buffer, encoded in our SBF compact binary format

function GetData(RecordBuffer: PUTF8Char; Field: TSynTableFieldProperties): pointer;
Retrieve the corresponding data address of a given field

function IterateJSONValues(Sender: TObject; Opaque: pointer; ID: integer; Data: pointer; DataLen: integer): boolean;
Can be used to retrieve all values matching a prepared TSynTableStatement
- this method matches the TSynBigTableIterateEvent callback definition
- Sender will be the TSynBigTable instance, and Opaque will point to a TSynTableStatement instance (with all fields initialized, including Writer)

function UpdateFieldEvent(Sender: TObject; Opaque: pointer; ID, Index: integer; Data: pointer; DataLen: integer): boolean;
This Event is to be called for all data records (via a GetIterating method) after any AddfieldUpdate, to refresh the data
- Opaque is in fact a pointer to a TUpdateFieldEvent record, and will contain all parameters set by TSynBigTableRecord.AddFieldUpdate, including a TFileBufferWriter instance to use to write the recreated data
- it will work with either any newly added field, handle also field data order change in SBF record (e.g. when a fixed-sized field has been added on a record containing variable-length fields)

function UpdateFieldRecord(RecordBuffer: PUTF8Char; var AvailableFields: TSQLFieldBits): TSBFString;
Update a record content after any AddfieldUpdate, to refresh the data
- AvailableFields must contain the list of existing fields in the previous data

function Validate(RecordBuffer: pointer; RecordIndex: integer): string;
Check the registered constraints according to a record SBF buffer
- returns "" on success
- returns an error message e.g. if a tftUnique constraint failed
- RecordIndex=-1 in case of adding, or the physical index of the updated record

procedure FieldIndexModify(aOldIndex, aNewIndex: integer; aOldRecordData, aNewRecordData: pointer);
Event which must be called by the storage engine when some values are modified
- if aOldIndex and aNewIndex are both >= 0, the corresponding aOldIndex will be replaced by aNewIndex value (i.e. called in case of a data Update)
- if aOldIndex is -1 and aNewIndex is >= 0, aNewIndex refers to a just created item (i.e. called in case of a data Add)
- if aOldIndex is >= 0 and aNewIndex is -1, aNewIndex refers to a just deleted item (i.e. called in case of a data Delete)
- will update then sort all existing TSynTableFieldProperties.OrderedIndex values
- the GetDataBuffer protected virtual method must have been overridden to properly return the record data for a given "physical/stored" index
- aOldRecordData and aNewRecordData can be specified in order to guess if the field data has really been modified (speed up the update a lot to only sort indexed fields if its content has been really modified)
**procedure** Filter(var RecordBuffer: TSBFString);

*Filter the SBF buffer record content with all registered filters*
- all field values are filtered in-place, following our SBF compact binary format encoding for this record

**procedure** GetJSONValues(aID: integer; RecordBuffer: PUTF8Char; W: TJSONWriter);

*Return the UTF-8 encoded JSON objects for the values contained in the specified RecordBuffer encoded in our SBF compact binary format, according to the Expand/WithID/Fields parameters of W*
- if W.Expand is true, JSON data is an object, for direct use with any Ajax or .NET client:
  ```json
  {"col1":val11,"col2":"val12"
  ```
- if W.Expand is false, JSON data is serialized (as used in TSQLTableJSON)
  ```json
  { "fieldCount":1,"values":["col1","col2",val11,"val12",val21,..] }
  ```
- only fields with a bit set in W.Fields will be appended
- if W.WithID is true, then the first ID field value is included

**procedure** LoadFrom(var RD: TFileBufferReader);

*Create a table definition instance from a specified file reader*

**procedure** SaveTo(WR: TFileBufferWriter);

*Save field properties to a specified file writer*

**procedure** UpdateFieldData(RecordBuffer: PUTF8Char; RecordBufferLen, FieldIndex: integer; var result: TSBFString; const NewFieldData: TSBFString='');

*Update a record content*
- return the updated record data, in our SBF compact binary format
- if NewFieldData is not specified, a default 0 or '' value is appended
- if NewFieldData is set, it must match the field value kind
- warning: this method will update result in-place, so RecordBuffer MUST be <> pointer(result) or data corruption may occur

**property** AddedField: TList read fAddedField write fAddedField;

*List of TSynTableFieldProperties added via all AddField() call*
- this list will allow TSynBigTableRecord.AddFieldUpdate to refresh the data on disk according to the new field configuration

**property** DefaultRecordData: TSBFString read fDefaultRecordData;

*Return a default content for ALL record fields*
- uses our SBF compact binary format

**property** Field[Index: integer]: TSynTableFieldProperties read GetFieldType;

*Retrieve the properties of a given field*
- returns nil if the specified Index is out of range

**property** FieldCount: integer read GetFieldCount;

*Number of fields in this table*

**property** FieldFromName[const aName: RawUTF8]: TSynTableFieldProperties read GetFieldFromName;

*Retrieve the properties of a given field*
- returns nil if the specified Index is out of range
property FieldIndexFromName[const aName: RawUTF8]: integer read GetFieldIndexFromName;

Retrieve the index of a given field
- returns -1 if the specified index is out of range

property FieldList: TObjectList read fField;

Read-only access to the Field list

property FieldVariableOffset: PtrUInt read fFieldVariableOffset;

Offset of the first variable length value field

property GetRecordData: TSynTableGetRecordData read fGetRecordData write fGetRecordData;

Event used for proper data retrieval of a given record buffer, according to the physical/storage index value (not per-ID index)
- if not set, field indexes won't work
- will be mapped e.g. to TSynBigTable.GetPointerFromPhysicalIndex

property HasUniqueIndexes: boolean read fFieldHasUniqueIndexes;

True if any field has a tfoUnique option set

property TableName: RawUTF8 read fTableName write fTableName;

The internal Table name used to identify it (e.g. from JSON or SQL)
- similar to the SQL Table name

TSynTableVariantType = class(TSynInvokeableVariantType)
A custom variant type used to have direct access to a record content
- use TSynTable.Data method to retrieve such a Variant
- this variant will store internally a SBF compact binary format representation of the record content
- uses internally a TSynTableData object

class function ToID(const V: Variant): integer;

Retrieve the ID value associated to a record content

class function ToSBF(const V: Variant): TSBFString;

Retrieve the SBF compact binary format representation of a record content

class function ToTable(const V: Variant): TSynTable;

Retrieve the TSynTable instance associated to a record content

procedure Clear(var V: TVarData); override;

Clear the content

procedure Copy(var Dest: TVarData; const Source: TVarData; const Indirect: Boolean); override;

Copy two record content

Types implemented in the SynTable unit

PRawByteStringGroup = ^TRawByteStringGroup;
Pointer reference to a TRawByteStringGroup

PSQLFieldBits = ^TSQLFieldBits;
Points to a bit set used for all available fields in a Table

PSynTableData = ^TSynTableData;
A pointer to structure used to store a TSynTable record

PSynUniqueIdentifierBits = ^TSynUniqueIdentifierBits;
Points to a 64-bit integer identifier, as computed by TSynUniqueIdentifierGenerator
- may be used to access the identifier internals, from its stored Int64 or TSynUniqueIdentifier value

PSynValidate = ^TSynValidate;
Points to a TSynValidate variable
- used e.g. as optional parameter to TSQLRecord.Validate/FilterAndValidate

TBlockingEvent = ( evNone, evWaiting, evTimeOut, evRaised );
The current state of a TBlockingProcess instance

TBlockingProcessEventPoolCall = type integer;
Used to identify each TBlockingProcessEventPool call
- allow to match a given TBlockingProcessEventPoolItem semaphore

TBlockingProcessEventPoolItemClass = class of TBlockingProcessEventPoolItem;
Class-reference type (metaclass) of a TBlockingProcess

TCompareOperator = ( soEqualTo, soNotEqualTo, soLessThan, soLessThanOrEqualTo, soGreaterThan, soGreaterThanOrEqualTo, soBeginWith, soContains, soSoundsLikeEnglish, soSoundsLikeFrench, soSoundsLikeSpanish );
SQL Query comparison operators
- used e.g. by CompareOperator() functions in SynTable.pas or vt_BestIndex() in mORMotSQLite3.pas

TConsoleColor = ( ccBlack, ccBlue, ccGreen, ccCyan, ccRed, ccMagenta, ccBrown, ccLightGray, ccDarkGray, ccLightBlue, ccLightGreen, ccLightCyan, ccLightRed, ccLightMagenta, ccYellow, ccWhite );
Available console colors (under Windows at least)

TDeltaError = ( dsSuccess, dsCrcCopy, dsCrcComp, dsCrcBegin, dsCrcEnd, dsCrcExtract, dsFlag, dsLen );
Result of function DeltaExtract()

TDiskPartitions = array of TDiskPartition;
Stores information about several disk partitions

Map the first Unicode page of Emojis, from U+1F600 to U+1F64F
- naming comes from github/Markdown :identifiers:

```pascal
TExprNodeType = (entWord, entNot, entOr, entAnd);
```

*Identify an expression search engine node type, as used by TExprParser*

```pascal
TExprNodeWordClass = class of TExprNodeWordAbstract;
```

*Class-reference type (metaclass) for a TExprNode*

- allow to customize the actual searching process for entWord

```pascal
TExprParserResult = (eprSuccess, eprNoExpression, eprMissingParenthesis,
eprTooManyParenthesis, eprMissingFinalWord, eprInvalidExpression, eprUnknownVariable,
eprUnsupportedOperator, eprInvalidConstantOrVariable);
```

*Results returned by TExprParserAbstract.Parse method*

```pascal
TFileBufferWriterKind = (wkUInt32, wkVarUInt32, wkVarInt32, wkSorted, wkOffsetU,
wkOffsetI, wkFakeMarker);
```

*Available kind of integer array storage, corresponding to the data layout*

- wkUInt32 will write the content as "plain" 4 bytes binary (this is the preferred way if the integers can be negative)
- wkVarUInt32 will write the content using our 32-bit variable-length integer encoding
- wkVarInt32 will write the content using our 32-bit variable-length integer encoding and the by-two complement (0=0,1=-1,2=2,4=-2...)
- wkSorted will write an increasing array of integers, handling the special case of a difference of similar value (e.g. 1) between two values - note that this encoding is efficient only if the difference is main < 253
- wkOffsetU and wkOffsetI will write the difference between two successive values, handling constant difference (Unsigned or Integer) in an optimized manner
- wkFakeMarker won't be used by WriteVarUInt32Array, but to notify a custom encoding

```pascal
TMatchDynArray = array of TMatch;
```

*Use SetMatches() to initialize such an array from a CSV pattern text*

```pascal
TNotifyThreadEvent = procedure(Sender: TThread) of object;
```

*Event prototype used e.g. by TSynBackgroundThreadAbstract callbacks*

- a similar signature is defined in SynCrtSock and LVCL.Classes

```pascal
TNullableBoolean = type variant;
```

*Define a variant published property as a nullable boolean*

- either a varNull or a varBoolean value will be stored in the variant
- either a NULL or a 0/1 INTEGER value will be stored in the database
- the property should be defined as such:

```pascal
property Bool: TNullableBoolean read fBool write fBool;
```

```pascal
TNullableCurrency = type variant;
```

*Define a variant published property as a nullable decimal value*

- either a varNull or a varCurrency value will be stored in the variant
- either a NULL or a FLOAT value will be stored in the database
- the property should be defined as such:

```pascal
property Cur: TNullableCurrency read fCur write fCur;
```

```pascal
TNullableDateTime = type variant;
```

*Define a variant published property as a nullable date/time value*

- SynTable.pas unit - Rev. 1.18

Page 1800 of 2552
- either a varNull or a varDate value will be stored in the variant
- either a NULL or a ISO-8601 TEXT value will be stored in the database
- the property should be defined as such:
  
  ```delphi
  property Dat: TNullableDateTime read fDat write fDat;
  ```

**TNullableFloat = type variant;**

*Define a variant published property as a nullable floating point value*

- either a varNull or a varDouble value will be stored in the variant
- either a NULL or a FLOAT value will be stored in the database
- the property should be defined as such:
  
  ```delphi
  property Flt: TNullableFloat read fFlt write fFlt;
  ```

**TNullableInteger = type variant;**

*Define a variant published property as a nullable integer*

- either a varNull or a varInt64 value will be stored in the variant
- either a NULL or an INTEGER value will be stored in the database
- the property should be defined as such:
  
  ```delphi
  property Int: TNullableInteger read fInt write fInt;
  ```

**TNullableTimeLog = type variant;**

*Define a variant published property as a nullable timestamp value*

- either a varNull or a varInt64 value will be stored in the variant
- either a NULL or a TTimeLog INTEGER value will be stored in the database
- the property should be defined as such:
  
  ```delphi
  property Tim: TNullableTimeLog read fTim write fTim;
  ```

**TNullableUTF8Text = type variant;**

*Define a variant published property as a nullable UTF-8 encoded text*

- either a varNull or varString (RawUTF8) will be stored in the variant
- either a NULL or a TEXT value will be stored in the database
- the property should be defined as such:
  
  ```delphi
  property Txt: TNullableUTF8Text read fTxt write fTxt;
  ```

or for a fixed-width VARCHAR (in external databases), here of 32 max chars:

```delphi
property Txt: TNullableUTF8Text index 32 read fTxt write fTxt;
```

- warning: prior to Delphi 2009, since the variant will be stored as RawUTF8 internally, you should not use directly the field value as a VCL string=AnsiString like string(aField) but use VariantToString(aField)

**TOnIdleSynBackgroundThread = procedure(Sender: TSynBackgroundThreadAbstract; ElapsedMS: Integer) of object;**

*Idle method called by TSynBackgroundThreadAbstract in the caller thread during remote blocking process in a background thread*

- typical use is to run Application.ProcessMessages, e.g. for TSQLRestClientURI.URI() to provide a responsive UI even in case of slow blocking remote access
- provide the time elapsed (in milliseconds) from the request start (can be used e.g. to popup a temporary message to wait)
- is call once with ElapsedMS=0 at request start
- is call once with ElapsedMS=1 at request ending
- see TLoginForm.OnIdleProcess and OnIdleProcessForm in mORMotUILogin.pas

```pascal
TOnProcessSynBackgroundThread = procedure(Sender: TSynBackgroundThreadEvent; ProcessOpaqueParam: pointer) of object;
  Background process method called by TSynBackgroundThreadEvent
  - will supply the OpaqueParam parameter as provided to RunAndWait() method when the Process virtual method will be executed

TOnProcessSynBackgroundThreadProc = procedure(ProcessOpaqueParam: pointer);
  Background process procedure called by TSynBackgroundThreadProcedure
  - will supply the OpaqueParam parameter as provided to RunAndWait() method when the Process virtual method will be executed

TOnSynBackgroundThreadProcess = procedure(Sender: TSynBackgroundThreadProcess; Event: TWaitResult) of object;
  Event callback executed periodically by TSynBackgroundThreadProcess
  - Event is wrTimeout after the OnProcessMS waiting period
  - Event is wrSignaled if ProcessEvent.SetEvent has been called

TOnSynBackgroundTimerProcess = procedure(Sender: TSynBackgroundTimer; Event: TWaitResult; const Msg: RawUTF8) of object;
  Event callback executed periodically by TSynBackgroundThreadProcess
  - Event is wrTimeout after the OnProcessMS waiting period
  - Event is wrSignaled if ProcessEvent.SetEvent has been called
  - Msg is "" if there is no pending message in this task FIFO
  - Msg is set for each pending message in this task FIFO

TOnSystemUseMeasured = procedure(ProcessID: integer; const Data: TSystemUseData) of object;
  Event handler which may be executed by TSystemUse.BackgroundExecute
  - called just after the measurement of each process CPU and RAM consumption
  - run from the background thread, so should not directly make VCL calls, unless BackgroundExecute is run from a VCL timer

TPendingTaskListItemDynArray = array of TPendingTaskListItem;
  Internal list definition, used by TPendingTaskList storage

TRawByteStringGroupValueDynArray = array of TRawByteStringGroupValue;
  Items as stored in a TRawByteStringGroup instance

TSBFString = type RawByteString;
  An custom RawByteString type used to store internally a data in our SBF compact binary format

TSQldbFieldType = ( ftUnknown, ftNull, ftInt64, ftDouble, ftCurrency, ftDate, ftUTF8, ftBlob );
  Handled field/parameter/column types for abstract database access
  - will map JSON-compatible low-level database-level access types, not high-level Delphi types as TSQldbField defined in mORMot.pas
  - it does not map either all potential types as defined in DB.pas (which are there for compatibility with old RDBMS, and are not abstract enough)
  - those types can be mapped to standard SQLite3 generic types, i.e. NULL, INTEGER, REAL, TEXT, BLOB (with the addition of a ftCurrency and ftDate type, for better support of most DB engines) see @http://www.sqlite.org/datatype3.html
  - the only string type handled here uses UTF-8 encoding (implemented using our RawUTF8 type), for
cross-Delphi true Unicode process

TSQLDBFieldTypeArray = array[0..MAX_SQLFIELDS-1] of TSQLDBFieldType;
  Array of field/parameter/column types for abstract database access
  - this array as a fixed size, ready to handle up to MAX_SQLFIELDS items

TSQLDBFieldTypeDynArray = array of TSQLDBFieldType;
  Array of field/parameter/column types for abstract database access

TSQLDBFieldTypes = set of TSQLDBFieldType;
  Set of field/parameter/column types for abstract database access

TSQLFieldBits = set of 0..MAX_SQLFIELDS-1;
  Used to store bit set for all available fields in a Table
  - with current MAX_SQLFIELDS value, 64 bits uses 8 bytes of memory
  - see also IsZero() and IsEqual() functions
  - you can also use ALL_FIELDS as defined in mORMot.pas

TSQLFieldIndex = SmallInt;
  Used to store a field index in a Table
  - note that -1 is commonly used for the ID/RowID field so the values should be signed
  - even if ShortInt (-128..127) may have been enough, we define a 16 bit safe unsigned integer to let
    the source compile with Delphi 5

TSQLFieldIndexDynArray = array of TSQLFieldIndex;
  -32768..32767 used to store field indexes in a Table
  - same as TSQLFieldBits, but allowing to store the proper order

TSQLParamType = ( sptUnknown, sptInteger, sptFloat, sptText, sptBlob, sptDateTime );
  Generic parameter types, as recognized by SQLParamContent() and ExtractInlineParameters() functions

TSQLParamTypeDynArray = array of TSQLParamType;
  Array of parameter types, as recognized by SQLParamContent() and ExtractInlineParameters() functions

TSQLVarDynArray = array of TSQLVar;
  Dynamic array of database values by reference storage

TSQLVarOption = ( svoDateWithMS );
  How TSQLVar may be processed
  - by default, ftDate will use seconds resolution unless svoDateWithMS is set

TSQLVarOptions = set of TSQLVarOption;
  Defines how TSQLVar may be processed

TSynAuthenticationClass = class of TSynAuthenticationAbstract;
  Class-reference type (metaclass) of an authentication class

TSynBackgroundThreadProcessStep = ( flagIdle, flagStarted, flagFinished, flagDestroying );
  State machine status of the TSynBackgroundThreadAbstract process

TSynBackgroundThreadProcessSteps = set of TSynBackgroundThreadProcessStep;
  State machine statuses of the TSynBackgroundThreadAbstract process
**TSynBackgroundTimerTaskDynArray** = array of TSynBackgroundTimerTask;

Stores TSynBackgroundTimer internal registration list

**TSynFilterClass** = class of TSynFilter;

Class-reference type (metaclass) of a record filter (transformation)

**TSynFilterOrValidateClass** = class of TSynFilterOrValidate;

Class-reference type (metaclass) for a TSynFilter or a TSynValidate

**TSynParallelProcessMethod** = procedure(IndexStart, IndexStop: integer) of object;

Callback implementing some parallelized process for TSynParallelProcess
- if 0<=IndexStart<=IndexStop, it should execute some process

**TSynSoundExPronunciation** = ( sndxEnglish, sndxFrench, sndxSpanish, sndxNone );

Available pronunciations for our fast Soundex implementation

**TSynTableFieldBits** = set of 0..63;

Used to store bit set for all available fields in a Table
- with current format, maximum field count is 64

**TSynTableFieldIndex** = function(const PropName: RawUTF8): integer of object;

Function prototype used to retrieve the index of a specified property name
- 'ID' is handled separately: here must be available only the custom fields

**TSynTableFieldOption** = ( tfoIndex, tfoUnique, tfoCaseInsensitive );

Available option types for a field property
- tfoIndex is set if an index must be created for this field
- tfoUnique is set if field values must be unique (if set, the tfoIndex will be always forced)
- tfoCaseInsensitive can be set to make no difference between 'a' and 'A' (by default, comparison is case-sensitive) - this option has an effect not only if tfoIndex or tfoUnique is set, but also for iterating search

**TSynTableFieldOptions** = set of TSynTableFieldOption;

Set of option types for a field

**TSynTableFieldType** = ( tftUnknown, tftBoolean, tftUInt8, tftUInt16, tftUInt24, tftInt32, tftInt64, tftCurrency, tftDouble, tftVarUInt32, tftVarInt32, tftVarUInt64, tftWinAnsi, tftUTF8, tftBlobInternal, tftBlobExternal, tftVarInt64 );

The available types for any TSynTable field property
- this is used in our so-called SBF compact binary format (similar to BSON or Protocol Buffers)
- those types are used for both storage and JSON conversion
- basic types are similar to SQLite3, i.e. Int64/Double/UTF-8/Blob
- storage can be of fixed size, or of variable length
- you can specify to use WinAnsi encoding instead of UTF-8 for string storage (it can use less space on disk than UTF-8 encoding)
- BLOB fields can be either internal (i.e. handled by TSynTable like a RawByteString text storage), either external (i.e. must be stored in a dedicated storage structure - e.g. another TSynBigTable instance)

**TSynTableFieldTypes** = set of TSynTableFieldType;

Set of available field types for TSynTable

**TSynTableGetRecordData** = function( Index: integer; var aTempData: RawByteString): pointer of object;

Function prototype used to retrieve the RECORD data of a specified Index
- the index is not the per-ID index, but the "physical" index, i.e. the index value used to retrieve data from low-level (and faster) method
- should return nil if Index is out of range
- caller must provide a temporary storage buffer to be used optionally

```pascal
TSynTableStatementOperator = ( opEqualTo, opNotEqualTo, opLessThan,
    opLessThanOrEqualTo, opGreaterThan, opGreaterThanOrEqualTo, opIn, opIsNull,
    opIsNotNull, opLike, opContains, opFunction );
```

The recognized operators for a TSynTableStatement where clause

```pascal
TSynTableStatementSelectDynArray = array of TSynTableStatementSelect;
```

The recognized SELECT expressions for TSynTableStatement

```pascal
TSynTableStatementWhereDynArray = array of TSynTableStatementWhere;
```

The recognized WHERE expressions for TSynTableStatement

```pascal
TSynUniqueIdentifier = type Int64;
```

64-bit integer unique identifier, as computed by TSynUniqueIdentifierGenerator
- they are increasing over time (so are much easier to store/shard/balance than UUID/GUID), and contain generation time and a 16-bit process ID
- mapped by TSynUniqueIdentifierBits memory structure
- may be used on client side for something similar to a MongoDB ObjectID, but compatible with TSQLRecord.ID: TID properties

```pascal
TSynUniqueIdentifierObfuscated = type RawUTF8;
```

A 24 chars cyphered hexadecimal string, mapping a TSynUniqueIdentifier
- has handled by TSynUniqueIdentifierGenerator.ToObfuscated/FromObfuscated

```pascal
TSynUniqueIdentifierProcess = type word;
```

16-bit unique process identifier, used to compute TSynUniqueIdentifier
- each TSynUniqueIdentifierGenerator instance is expected to have its own unique process identifier, stored as a 16 bit integer 1..65535 value

```pascal
TSystemUseDataDynArray = array of TSystemUseData;
```

Store CPU and RAM usage history for a given process
- as returned by TSystemUse.History

```pascal
TSystemUseProcessDynArray = array of TSystemUseProcess;
```

Internal storage of CPU and RAM usage for a set of processes

```pascal
TTextWriterHTMLEscape = set of ( heHtmlEscape, heEmojiToUTF8);
```

Tune AddHtmlEscapeWiki/AddHtmlEscapeMarkdown wrapper functions process
- heHtmlEscape will escape any HTML special chars, e.g. & into &amp;
- heEmojiToUTF8 will convert any Emoji text into UTF-8 Unicode character, recognizing e.g. :joy: or :) in the text

```pascal
TT imeZoneDataDynArray = array of TTimeZoneData;
```

Used to store the Time Zone information of a TSynTimeZone class

```pascal
TT imeZoneID = type RawUTF8;
```

Text identifier of a Time Zone, following Microsoft Windows naming

**Constants implemented in the SynTable unit**

```pascal
DELTA_BUF_DEFAULT = 2 shl 20;
```
2MB as internal chunks/window default size for DeltaCompress()
- will use up to 9 MB of RAM during DeltaCompress() - none in DeltaExtract()

DELTA_LEVEL_BEST = 500;

*Brutal pattern search depth for DeltaCompress()*
- may become very slow, with minor benefit, on huge content

DELTA_LEVEL_FAST = 100;

*Normal pattern search depth for DeltaCompress()*
- gives good results on most content

PARSER_STOPCHAR = ['&', '+', '-', '(' ')

*May be used when overriding TExprParserAbstract.ParseNextWord method*

SOUNDEX_BITS = 4;

*Number of bits to use for each interesting soundex char*
- default is to use 8 bits, i.e. 4 soundex chars, which is the standard approach
- for a more detailed soundex, use 4 bits resolution, which will compute up to 7 soundex chars in a cardinal (that's our choice)

SQLDBFIELDTYPE_TO_DELPHITYPE: array[TSQLDBFieldType] of RawUTF8 = ('?', ',', 'Int64', 'Double', 'Currency', 'TDateTime', 'RawUTF8', 'TSQLRawBlob');

*Convert identified field types into high-level ORM types*
- as will be implemented in unit mORMot.pas

SYNTABLESTATEMENTWHEREID = 0;

*Used by TSynTableStatement.WhereField for "SELECT .. FROM TableName WHERE ID=?"*

### Functions or procedures implemented in the SynTable unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddFieldIndex</td>
<td>Add a field index to an array of field indexes</td>
<td>1810</td>
</tr>
<tr>
<td>AddHtmlEscapeMarkdown</td>
<td>Convert minimal Markdown text into proper HTML</td>
<td>1810</td>
</tr>
<tr>
<td>AddHtmlEscapeWiki</td>
<td>Convert some wiki-like text into proper HTML</td>
<td>1810</td>
</tr>
<tr>
<td>CompareOperator</td>
<td>Low-level text comparison according to a specified operator</td>
<td>1811</td>
</tr>
<tr>
<td>CompareOperator</td>
<td>Low-level floating-point comparison according to a specified operator</td>
<td>1811</td>
</tr>
<tr>
<td>CompareOperator</td>
<td>Low-level integer comparison according to a specified operator</td>
<td>1811</td>
</tr>
<tr>
<td>ConsoleKeyPressed</td>
<td>Low-level access to the keyboard state of a given key</td>
<td>1811</td>
</tr>
<tr>
<td>ConsoleReadBody</td>
<td>Read all available content from stdin</td>
<td>1811</td>
</tr>
<tr>
<td>ConsoleShowFatalException</td>
<td>Could be used in the main program block of a console application to handle unexpected fatal exceptions</td>
<td>1811</td>
</tr>
<tr>
<td>ConsoleWaitForKey</td>
<td>Will wait for the ENTER key to be pressed, processing Synchronize() pending notifications, and the internal Windows Message loop (on this OS)</td>
<td>1811</td>
</tr>
<tr>
<td>ConsoleWrite</td>
<td>Write some text to the console using a given color</td>
<td>1812</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>ConsoleWrite</td>
<td>Write some text to the console using a given color</td>
<td>1812</td>
</tr>
<tr>
<td>DateTimeToSQL</td>
<td>Convert a date/time to a ISO-8601 string format for SQL '?' inlined parameters</td>
<td>1812</td>
</tr>
<tr>
<td>DateTimeToSQL</td>
<td>Convert a date to a ISO-8601 string format for SQL '?' inlined parameters</td>
<td>1812</td>
</tr>
<tr>
<td>DateToSQL</td>
<td>Convert a date to a ISO-8601 string format for SQL '?' inlined parameters</td>
<td>1812</td>
</tr>
<tr>
<td>DeltaCompress</td>
<td>Compute difference of two binary buffers</td>
<td>1813</td>
</tr>
<tr>
<td>DeltaCompress</td>
<td>Compute difference of two binary buffers</td>
<td>1813</td>
</tr>
<tr>
<td>DeltaCompress</td>
<td>Compute difference of two binary buffers</td>
<td>1813</td>
</tr>
<tr>
<td>DeltaExtract</td>
<td>Low-level apply the delta binary as computed by DeltaCompress()</td>
<td>1813</td>
</tr>
<tr>
<td>DeltaExtract</td>
<td>Apply the delta binary as computed by DeltaCompress()</td>
<td>1813</td>
</tr>
<tr>
<td>DeltaExtractSize</td>
<td>Returns how many bytes a DeltaCompress() result will expand to</td>
<td>1813</td>
</tr>
<tr>
<td>DeltaExtractSize</td>
<td>Returns how many bytes a DeltaCompress() result will expand to</td>
<td>1813</td>
</tr>
<tr>
<td>EmojiFromDots</td>
<td>Low-level conversion of github/Markdown :identifiers: into UTF-8 Emoji sequences</td>
<td>1813</td>
</tr>
<tr>
<td>EmojiFromDots</td>
<td>Conversion of github/Markdown :identifiers: into UTF-8 Emoji sequences</td>
<td>1813</td>
</tr>
<tr>
<td>EmojiFromText</td>
<td>Recognize github/Markdown compatible text of Emojis</td>
<td>1813</td>
</tr>
<tr>
<td>EmojiParseDots</td>
<td>Low-level parser of github/Markdown compatible text of Emojis</td>
<td>1813</td>
</tr>
<tr>
<td>EmojiToDots</td>
<td>Conversion of UTF-8 emoji sequences into github/Markdown :identifiers:</td>
<td>1814</td>
</tr>
<tr>
<td>EmojiToDots</td>
<td>Low-level conversion of UTF-8 emoji sequences into github/Markdown :identifiers:</td>
<td>1814</td>
</tr>
<tr>
<td>EnumAllProcesses</td>
<td>A wrapper around EnumProcesses() PsAPI call</td>
<td>1814</td>
</tr>
<tr>
<td>EnumProcessName</td>
<td>A wrapper around QueryFullProcessImageNameW/GetModuleFileNameEx PsAPI call</td>
<td>1814</td>
</tr>
<tr>
<td>ExtractInlineParameters</td>
<td>This function will extract inlined :{(1234)}: parameters into Types[]/Values[]</td>
<td>1814</td>
</tr>
<tr>
<td>FieldBitsToIndex</td>
<td>Convert a TSQLFieldBits set of bits into an array of integers</td>
<td>1814</td>
</tr>
<tr>
<td>FieldBitsToIndex</td>
<td>Convert a TSQLFieldBits set of bits into an array of integers</td>
<td>1814</td>
</tr>
<tr>
<td>FieldIndexToBits</td>
<td>Convert an array of field indexes into a TSQLFieldBits set of bits</td>
<td>1814</td>
</tr>
<tr>
<td>FieldIndexToBits</td>
<td>Convert an array of field indexes into a TSQLFieldBits set of bits</td>
<td>1814</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>FillZero</td>
<td>Fast initialize a TSQLFieldBits with 0</td>
<td>1814</td>
</tr>
<tr>
<td>FilterMatches</td>
<td>Apply the CSV-supplied glob patterns to an array of RawUTF8</td>
<td>1814</td>
</tr>
<tr>
<td>FixedWaitFor</td>
<td>Allow to fix TEvent.WaitFor() method for Kylix</td>
<td>1814</td>
</tr>
<tr>
<td>FixedWaitForever</td>
<td>Allow to fix TEvent.WaitFor(Event,INFINITE) method for Kylix</td>
<td>1814</td>
</tr>
<tr>
<td>GetDiskInfo</td>
<td>Retrieve low-level information about a given disk partition</td>
<td>1815</td>
</tr>
<tr>
<td>GetDiskPartitions</td>
<td>Retrieve low-level information about all mounted disk partitions of the system</td>
<td>1815</td>
</tr>
<tr>
<td>GetDiskPartitionsText</td>
<td>Retrieve low-level information about all mounted disk partitions as text</td>
<td>1815</td>
</tr>
<tr>
<td>GetMemoryInfo</td>
<td>Retrieve low-level information about current memory usage</td>
<td>1815</td>
</tr>
<tr>
<td>HtmlEscapeMarkdown</td>
<td>Escape some Markdown-marked text into HTML</td>
<td>1815</td>
</tr>
<tr>
<td>HtmlEscapeWiki</td>
<td>Escape some wiki-marked text into HTML</td>
<td>1815</td>
</tr>
<tr>
<td>InlineParameter</td>
<td>Returns a string value as inlined '(:&quot;value&quot;:)' text</td>
<td>1815</td>
</tr>
<tr>
<td>InlineParameter</td>
<td>Returns a 64-bit value as inlined ':(1234):' text</td>
<td>1815</td>
</tr>
<tr>
<td>IPToCardinal</td>
<td>Convert an IPv4 'x.x.x.x' text into its 32-bit value</td>
<td>1816</td>
</tr>
<tr>
<td>IPToCardinal</td>
<td>Convert an IPv4 'x.x.x.x' text into its 32-bit value, 0 or localhost</td>
<td>1816</td>
</tr>
<tr>
<td>isEqual</td>
<td>Fast comparison of two TSQLFieldBits values</td>
<td>1816</td>
</tr>
<tr>
<td>IsMatch</td>
<td>Return TRUE if the supplied content matches a glob pattern</td>
<td>1816</td>
</tr>
<tr>
<td>IsMatchString</td>
<td>Return TRUE if the supplied content matches a glob pattern, using VCL strings</td>
<td>1816</td>
</tr>
<tr>
<td>Iso8601ToSQL</td>
<td>Convert a Iso8601 encoded string into a ISO-8601 string format for SQL '?' inlined parameters</td>
<td>1816</td>
</tr>
<tr>
<td>IsValidEmail</td>
<td>Return TRUE if the supplied content is a valid email address</td>
<td>1816</td>
</tr>
<tr>
<td>IsValidIP4Address</td>
<td>Return TRUE if the supplied content is a valid IP v4 address</td>
<td>1816</td>
</tr>
<tr>
<td>IsZero</td>
<td>Returns TRUE if no bit inside this TSQLFieldBits is set</td>
<td>1816</td>
</tr>
<tr>
<td>MatchAdd</td>
<td>Add one TMach if not already registered in the Several[] dynamic array</td>
<td>1817</td>
</tr>
<tr>
<td>MatchAny</td>
<td>Returns TRUE if Match=nil or if any Match[].Match(Text) is TRUE</td>
<td>1817</td>
</tr>
<tr>
<td>MatchExists</td>
<td>Search if one TMach is already registered in the Several[] dynamic array</td>
<td>1817</td>
</tr>
<tr>
<td>NullableBoolean</td>
<td>Creates a nullable Boolean value from a supplied constant</td>
<td>1817</td>
</tr>
<tr>
<td>NullableBooleanIsEmptyOrNull</td>
<td>Same as VarIsEmpty(V) or VarIsEmpty(V), but faster</td>
<td>1817</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>NullableBooleanToValue</td>
<td>Check if a TNullableBoolean is null, or return its value</td>
<td>1817</td>
</tr>
<tr>
<td>NullableBooleanToValue</td>
<td>Check if a TNullableBoolean is null, or return its value</td>
<td>1817</td>
</tr>
<tr>
<td>NullableCurrency</td>
<td>Creates a nullable Currency value from a supplied constant</td>
<td>1817</td>
</tr>
<tr>
<td>NullableCurrencyIsEmptyOrNull</td>
<td>Same as VarIsEmpty(V) or VarIsEmpty(V), but faster</td>
<td>1817</td>
</tr>
<tr>
<td>NullableCurrencyToValue</td>
<td>Check if a TNullableCurrency is null, or return its value</td>
<td>1817</td>
</tr>
<tr>
<td>NullableCurrencyToValue</td>
<td>Check if a TNullableCurrency is null, or return its value</td>
<td>1817</td>
</tr>
<tr>
<td>NullableDateTime</td>
<td>Creates a nullable TDateTime value from a supplied constant</td>
<td>1817</td>
</tr>
<tr>
<td>NullableDateTimeIsEmptyOrNull</td>
<td>Same as VarIsEmpty(V) or VarIsEmpty(V), but faster</td>
<td>1818</td>
</tr>
<tr>
<td>NullableDateTimeToValue</td>
<td>Check if a TNullableDateTime is null, or return its value</td>
<td>1818</td>
</tr>
<tr>
<td>NullableDateTimeToValue</td>
<td>Check if a TNullableDateTime is null, or return its value</td>
<td>1818</td>
</tr>
<tr>
<td>NullableFloat</td>
<td>Creates a nullable floating-point value from a supplied constant</td>
<td>1818</td>
</tr>
<tr>
<td>NullableFloatIsEmptyOrNull</td>
<td>Same as VarIsEmpty(V) or VarIsEmpty(V), but faster</td>
<td>1818</td>
</tr>
<tr>
<td>NullableFloatToValue</td>
<td>Check if a TNullableFloat is null, or return its value</td>
<td>1818</td>
</tr>
<tr>
<td>NullableFloatToValue</td>
<td>Check if a TNullableFloat is null, or return its value</td>
<td>1818</td>
</tr>
<tr>
<td>NullableInteger</td>
<td>Creates a nullable integer value from a supplied constant</td>
<td>1818</td>
</tr>
<tr>
<td>NullableIntegerIsEmptyOrNull</td>
<td>Same as VarIsEmpty(V) or VarIsEmpty(V), but faster</td>
<td>1818</td>
</tr>
<tr>
<td>NullableIntegerToValue</td>
<td>Check if a TNullableInteger is null, or return its value</td>
<td>1818</td>
</tr>
<tr>
<td>NullableIntegerToValue</td>
<td>Check if a TNullableInteger is null, or return its value</td>
<td>1818</td>
</tr>
<tr>
<td>NullableTimeLog</td>
<td>Creates a nullable TTimeLog value from a supplied constant</td>
<td>1819</td>
</tr>
<tr>
<td>NullableTimeLogIsEmptyOrNull</td>
<td>Same as VarIsEmpty(V) or VarIsEmpty(V), but faster</td>
<td>1819</td>
</tr>
<tr>
<td>NullableTimeLogToValue</td>
<td>Check if a TNullableTimeLog is null, or return its value</td>
<td>1819</td>
</tr>
<tr>
<td>NullableTimeLogToValue</td>
<td>Check if a TNullableTimeLog is null, or return its value</td>
<td>1819</td>
</tr>
<tr>
<td>NullableUTF8Text</td>
<td>Creates a nullable UTF-8 encoded text value from a supplied constant</td>
<td>1819</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>NullableUTF8TextIsEmptyOrNull</td>
<td>Same as VarIsEmpty(V) or VarIsEmpty(V), but faster</td>
<td>1819</td>
</tr>
<tr>
<td>NullableUTF8TextToValue</td>
<td>Check if a TNullableUTF8Text is null, or return its value</td>
<td>1819</td>
</tr>
<tr>
<td>NullableUTF8TextToValue</td>
<td>Check if a TNullableUTF8Text is null, or return its value</td>
<td>1819</td>
</tr>
<tr>
<td>SearchFieldIndex</td>
<td>Search a field index in an array of field indexes</td>
<td>1819</td>
</tr>
<tr>
<td>SetMatches</td>
<td>Fill the Match[0..MatchMax] static array with all glob patterns supplied as CSV</td>
<td>1820</td>
</tr>
<tr>
<td>SetMatches</td>
<td>Fill the Match[] dynamic array with all glob patterns supplied as CSV</td>
<td>1820</td>
</tr>
<tr>
<td>SoundExAnsi</td>
<td>Retrieve the Soundex value of a text word, from Ansi buffer</td>
<td>1820</td>
</tr>
<tr>
<td>SoundExAnsi</td>
<td>Retrieve the Soundex value of a text word, from Ansi buffer</td>
<td>1820</td>
</tr>
<tr>
<td>SoundExUTF8</td>
<td>Retrieve the Soundex value of a text word, from UTF-8 buffer</td>
<td>1820</td>
</tr>
<tr>
<td>SQLParamContent</td>
<td>Guess the content type of an UTF-8 SQL value, in :(...): format</td>
<td>1820</td>
</tr>
<tr>
<td>SQLToDateTime</td>
<td>Decode a SQL '?' inlined parameter (i.e. with JSON_SQLDATE_MAGIC prefix)</td>
<td>1820</td>
</tr>
<tr>
<td>SQLVarLength</td>
<td>Returns the stored size of a TSQLVar database value</td>
<td>1821</td>
</tr>
<tr>
<td>StringToConsole</td>
<td>Direct conversion of a VCL string into a console OEM-encoded String</td>
<td>1821</td>
</tr>
<tr>
<td>SynTableVariantVarTyp e</td>
<td>Initialize TSynTableVariantType if needed, and return the corresponding VType</td>
<td>1821</td>
</tr>
<tr>
<td>SystemInfoJson</td>
<td>Returns a JSON object containing basic information about the computer</td>
<td>1821</td>
</tr>
<tr>
<td>TextBackgroundColor</td>
<td>Change the console text background color</td>
<td>1821</td>
</tr>
<tr>
<td>TextColor</td>
<td>Change the console text writing color</td>
<td>1821</td>
</tr>
<tr>
<td>TextToSQLDBFieldTyp e</td>
<td>Guess the correct TSQLDBFieldType from the UTF-8 representation of a value</td>
<td>1821</td>
</tr>
<tr>
<td>TimeLogToSQL</td>
<td>Convert a TTimeLog value into a ISO-8601 string format for SQL '?' inlined parameters</td>
<td>1821</td>
</tr>
<tr>
<td>ToSBFFStr</td>
<td>Convert any AnsiString content into our SBF compact binary format storage</td>
<td>1821</td>
</tr>
<tr>
<td>Utf8ToConsole</td>
<td>Direct conversion of a UTF-8 encoded string into a console OEM-encoded String</td>
<td>1821</td>
</tr>
<tr>
<td>VariantToSQLVar</td>
<td>Convert any Variant into a database value</td>
<td>1822</td>
</tr>
<tr>
<td>VariantTypeToSQLDBFieldTyp e</td>
<td>Guess the correct TSQLDBFieldType from a variant value</td>
<td>1822</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>VariantVTTypeToSQLDBFieldtype</td>
<td>Guess the correct TSQldbFieldtype from a variant type</td>
<td>1822</td>
</tr>
<tr>
<td>ZeroCompress</td>
<td>RLE compression of a memory buffer containing mostly zeros</td>
<td>1822</td>
</tr>
<tr>
<td>ZeroCompressXor</td>
<td>RLE compression of XORed memory buffers resulting in mostly zeros</td>
<td>1822</td>
</tr>
<tr>
<td>ZeroDecompress</td>
<td>RLE uncompression of a memory buffer containing mostly zeros</td>
<td>1822</td>
</tr>
<tr>
<td>ZeroDecompressOr</td>
<td>RLE uncompression and ORing of a memory buffer containing mostly zeros</td>
<td>1822</td>
</tr>
</tbody>
</table>

**function** AddFieldIndex(var Indexes: TSQLFieldIndexDynArray; Field: integer): integer;

*Add a field index to an array of field indexes*
- returns the index in Indexes[] of the newly appended Field value

**procedure** AddHtmlEscapeMarkdown(W: TTextWriter; P: PUTF8Char; esc: TTextWriterHTMLEscape=[heEmojiToUTF8]);

*Convert minimal Markdown text into proper HTML*
- convert all #13#10 into `<p>...</p>`, `*..*` into `<em>..</em>`, `**..**` into `<strong>..</strong>`, `\` into `<code>...</code>`, backslash espaces `\\` and so on, [title](http://...) and detect plain http:// as `<a href=...>`
- create unordered lists from trailing `* ` chars, blockquotes from trailing `>` char, and code line from 4 initial spaces
- as with default Markdown, won’t escape HTML special chars (i.e. you can write plain HTML in the supplied text) unless esc is set otherwise
- only inline-style links and images are supported yet (not reference-style); tables aren’t supported either

**procedure** AddHtmlEscapeWiki(W: TTextWriter; P: PUTF8Char; esc: TTextWriterHTMLEscape=[heHTMLEscape,heEmojiToUTF8]);

*Convert some wiki-like text into proper HTML*
- convert all #13#10 into `<p>...</p>`, `*..*` into `<em>..</em>`, `+..+` into `<strong>..</strong>`, `\` into `<code>...</code>`, and http://... as `<a href=http://...>`
- escape any HTML special chars, and Emoji tags as specified with esc

**function** CompareOperator(FieldType: TSynTableFieldType; SBF, SBFEnd: PUTF8Char; Value: Int64; Oper: TCompareOperator): boolean; overload;

*Low-level integer comparison according to a specified operator*
- SBF must point to the values encoded in our SBF compact binary format
- Value must contain the plain integer value
- Value can be a Currency accessed via a PInt64
- will work only for tftBoolean, tftUInt8, tftUInt16, tftUInt24, tftInt32, tftInt64 and tftCurrency field types
- will handle only soEqualTo...soGreaterThanOrEqualTo operators
- if SBFEnd is not nil, it will test for all values until SBF>=SBFEnd (can be used for tftArray)
- returns true if both values match, or false otherwise
function CompareOperator(SBF, SBFEnd: PUTF8Char; Value: double; Oper: TCompareOperator): boolean; overload;

Low-level floating-point comparison according to a specified operator

- SBF must point to the values encoded in our SBF compact binary format
- Value must contain the plain floating-point value
- will work only for tftDouble field type
- will handle only soEqualTo...soGreaterThanOrEqualTo operators
- if SBFEnd is not nil, it will test for all values until SBF>=SBE
  nd (can be used for tftArray)
- returns true if both values match, or false otherwise

function CompareOperator(FieldType: TSynTableFieldType; SBF, SBFEnd: PUTF8Char; Value: PUTF8Char; ValueLen: integer; Oper: TCompareOperator; CaseSensitive: boolean): boolean; overload;

Low-level text comparison according to a specified operator

- SBF must point to the values encoded in our SBF compact binary format
- Value must contain the plain text value, in the same encoding (either WinAnsi either UTF-8, as
  FieldType defined for the SBF value)
- will work only for tftWinAnsi and tftUTF8 field types
- will handle all kind of operators - including soBeginWith, soContains and soSoundsLike* - but
  soSoundsLike* won't make use of the CaseSensitive parameter
- for soSoundsLikeEnglish, soSoundsLikeFrench and soSoundsLikeSpanish operators, Value is not a
  real PUTF8Char but a prepared PSynSoundEx
- if SBFEnd is not nil, it will test for all values until SBF>=SBE
  nd (can be used for tftArray)
- returns true if both values match, or false otherwise

function ConsoleKeyPressed(ExpectedKey: Word): Boolean;

Low-level access to the keyboard state of a given key

function ConsoleReadBody: RawByteString;

Read all available content from stdin

- could be used to retrieve some file piped to the command line
- the content is not converted, so will follow the encoding used for storage

procedure ConsoleShowFatalException(E: Exception; WaitForEnterKey: boolean=true);

Could be used in the main program block of a console application to handle unexpected fatal
exceptions

- typical use may be:
begin
  try
    ... // main console process
  except
    on E: Exception do
    begin
      ConsoleShowFatalException(E);
      ConsoleShowFatalException(E);
    end;
  end.
end.

procedure ConsoleWaitForEnterKey;

Will wait for the ENTER key to be pressed, processing Synchronize() pending notifications, and the
internal Windows Message loop (on this OS)

- to be used e.g. for proper work of console applications with interface-based service
  implemented as optExecInMainThread
procedure ConsoleWrite(const Fmt: RawUTF8; const Args: array of const; Color: TConsoleColor=ccLightGray; NoLineFeed: boolean=false); overload;
  Write some text to the console using a given color

procedure ConsoleWrite(const Text: RawUTF8; Color: TConsoleColor=ccLightGray; NoLineFeed: boolean=false; NoColor: boolean=false); overload;
  Write some text to the console using a given color

function DateTimeToSQL(DT: TDateTime; WithMS: boolean=false): RawUTF8;
  Convert a date/time to a ISO-8601 string format for SQL '?' inlined parameters
- if DT=0, returns ""
- if DT contains only a date, returns the date encoded as '{0YYYY-MM-DD}'
- if DT contains only a time, returns the time encoded as '{0Thh:mm:ss}'
  (JSON_SQLDATE_MAGIC will be used as prefix to create '{...}' pattern)
- otherwise, returns the ISO-8601 date and time encoded as '{0YYYY-MM-DDThh:mm:ss}'
  (JSON_SQLDATE_MAGIC will be used as prefix to create '{...}' pattern)
- if WithMS is TRUE, will append '.sss' for milliseconds resolution
- to be used e.g. as in:
  aRec.CreateAndFillPrepare(Client, 'Datum=?',[DateTimeToSQL(Now)]);
- see TimeLogToSQL() if you are using TTimeLog/TModTime/TCreateTime values

function DateToSQL(Year,Month,Day: cardinal): RawUTF8; overload;
  Convert a date to a ISO-8601 string format for SQL '?' inlined parameters
- will return the date encoded as '{0YYYY-MM-DD}'
  - therefore: '{20120504}': pattern will be recognized as a sftDateTime inline parameter in
    SQLParamContent()/ExtractInlineParameters() functions (JSON_SQLDATE_MAGIC will be used as prefix to create '{...}' pattern)
- to be used e.g. as in:
  aRec.CreateAndFillPrepare(Client, 'Datum=?',[DateToSQL(2012,5,4)]);

function DateToSQL(Date: TDateTime): RawUTF8; overload;
  Convert a date to a ISO-8601 string format for SQL '?' inlined parameters
- will return the date encoded as '{0YYYY-MM-DD}'
  - therefore: '{20120504}': pattern will be recognized as a sftDateTime inline parameter in
    SQLParamContent()/ExtractInlineParameters() functions (JSON_SQLDATE_MAGIC will be used as prefix to create '{...}' pattern)
- to be used e.g. as in:
  aRec.CreateAndFillPrepare(Client, 'Datum=?',[DateToSQL(EncodeDate(2012,5,4))]);

function DeltaCompress(const New, Old: RawByteString; Level: integer=DELTA_LEVEL_FAST; BufSize: integer=DELTA_BUF_DEFAULT): RawByteString; overload;
  Compute difference of two binary buffers
- returns '=' for equal buffers, or an optimized binary delta
- DeltaExtract() could be used later on to compute New from Old + Delta
```pascal
function DeltaCompress(New, Old: PAnsiChar; NewSize, OldSize: integer; out Delta: PAnsiChar; Level: integer=DELTA_LEVEL_FAST; BufSize: integer=DELTA_BUF_DEFAULT): integer; overload;
  Compute difference of two binary buffers
  - returns '=' for equal buffers, or an optimized binary delta
  - DeltaExtract() could be used later on to compute New from Old + Delta
  - caller should call Freemem(Delta) once finished with the output buffer

function DeltaCompress(New, Old: PAnsiChar; NewSize, OldSize: integer; Level: integer=DELTA_LEVEL_FAST; BufSize: integer=DELTA_BUF_DEFAULT): RawByteString; overload;
  Compute difference of two binary buffers
  - returns '=' for equal buffers, or an optimized binary delta
  - DeltaExtract() could be used later on to compute New from Old

function DeltaExtract(const Delta, Old: RawByteString; out New: RawByteString): TDeltaError; overload;
  Apply the delta binary as computed by DeltaCompress()
  - decompression don't use any RAM, will perform crc32c check, and is very fast
  - return dsSuccess if was uncompressed to aOutUpd as expected

function DeltaExtract(Delta, Old, New: PAnsiChar): TDeltaError; overload;
  Low-level apply the delta binary as computed by DeltaCompress()
  - New should already be allocated with DeltaExtractSize(Delta) bytes
  - as such, expect Delta, Old and New to be <> nil, and Delta <> '='
  - return dsSuccess if was uncompressed to aOutUpd as expected

function DeltaExtractSize(const Delta: RawByteString): integer; overload;
  Returns how many bytes a DeltaCompress() result will expand to

function DeltaExtractSize(Delta: PAnsiChar): integer; overload;
  Returns how many bytes a DeltaCompress() result will expand to

function EmojiFromDots(const text: RawUTF8): RawUTF8; overload;
  Conversion of github/Markdown :identifiers: into UTF-8 Emoji sequences

procedure EmojiFromDots(P: PUTF8Char; W: TTextWriter); overload;
  Low-level conversion of github/Markdown :identifiers: into UTF-8 Emoji sequences

function EmojiFromText(P: PUTF8Char; len: PtrInt): TEmoji;
  Recognize github/Markdown compatible text of Emojis
  - for instance 'sunglasses' text buffer will return eSunglasses
  - returns eNone if no case-insensitive match was found

function EmojiParseDots(var P: PUTF8Char; W: TTextWriter=nil): TEmoji;
  Low-level parser of github/Markdown compatible text of Emojis
  - supplied P^ should point to ':'
  - will append the recognized UTF-8 Emoji if P contains e.g. :joy: or :)
  - will append ':' if no Emoji text is recognized, and return eNone
  - will try both EMOJI_AFTERDOTS[] and EMOJI_RTTI[] reference set
  - if W is nil, won't append anything, but just return the recognized TEmoji

procedure EmojiToDots(P: PUTF8Char; W: TTextWriter); overload;
  Low-level conversion of UTF-8 Emoji sequences into github/Markdown :identifiers:
```

function EmojiToDots(const text: RawUTF8): RawUTF8; overload;

Conversion of UTF-8 Emoji sequences into github/Markdown identifiers:

function EnumAllProcesses(out Count: Cardinal): TCardinalDynArray;

A wrapper around EnumProcesses() PsAPI call

function EnumProcessName(PID: Cardinal): RawUTF8;

A wrapper around QueryFullProcessImageNameW/GetModuleFileNameEx PsAPI call

function ExtractInlineParameters(const SQL: RawUTF8; var Types: TSQLParamTypeDynArray; var Values: TRawUTF8DynArray; var maxParam: integer; var Nulls: TSQLFieldBits): RawUTF8;

This function will extract inlined :(1234): parameters into Types[/Values[]
- will return the generic SQL statement with ? place holders for inlined parameters and setting Values with SQLParamContent() decoded content
- will set maxParam=0 in case of no inlined parameters
- recognized types are sptInteger, sptFloat, sptDateTime (\uFFF1...'), sptUTF8Text and sptBlob (\uFFFO...')
- sptUnknown is returned on invalid content

function FieldBitsToIndex(const Fields: TSQLFieldBits; MaxLength: integer=MAX_SQLFIELDS): TSQLFieldIndexDynArray; overload;

Convert a TSQLFieldBits set of bits into an array of integers

procedure FieldIndexToBits(const Index: TSQLFieldIndexDynArray): TSQLFieldBits;

Convert an array of field indexes into a TSQLFieldBits set of bits

procedure FillZero(var Fields: TSQLFieldBits); overload;

Fast initialize a TSQLFieldBits with 0
- is optimized for 64, 128, 192 and 256 max bits count (i.e. MAX_SQLFIELDS)
- will work also with any other value

procedure FilterMatchs(const CSVPattern: RawUTF8; CaseInsensitive: boolean; var Values: TRawUTF8DynArray);

Apply the CSV-supplied glob patterns to an array of RawUTF8
- any text not maching the pattern will be deleted from the array

function FixedWaitFor(Event: TEvent; Timeout: LongWord): TWaitResult;

Allow to fix TEvent.WaitFor() method for Kylix
- under Windows or with FPC, will call original TEvent.WaitFor() method

procedure FixedWaitForever(Event: TEvent);

Allow to fix TEvent.WaitFor(Event,INFINITE) method for Kylix
- under Windows or with FPC, will call original TEvent.WaitFor() method
function GetDiskInfo(var aDriveFolderOrFile: TFileName; out aAvailableBytes, aFreeBytes, aTotalBytes: QWord; aVolumeName: PFileName = nil): boolean;

Retrieve low-level information about a given disk partition
- as used by TSynMonitorDisk and GetDiskPartitionsText()
- only under Windows the Quotas are applied separately to aAvailableBytes in respect to global aFreeBytes

function GetDiskPartitions: TDiskPartitions;

Retrieve low-level information about all mounted disk partitions of the system
- returned partitions array is sorted by "mounted" ascending order

function GetDiskPartitionsText(nocache: boolean=false; withfreespace: boolean=false; nospace: boolean=false): RawUTF8;

Retrieve low-level information about all mounted disk partitions as text
- returns e.g. under Linux '/dev/sda3 (19 GB), /boot /dev/sda2 (486.8 MB), /home /dev/sda4 (0.9 TB)' or under Windows 'C:\ System (115 GB), D:\ Data (99.3 GB)'
- uses internally a cache unless nocache is true
- includes the free space if withfreespace is true - e.g. '(80 GB / 115 GB)'

function GetMemoryInfo(out info: TMemoryInfo; withalloc: boolean): boolean;

Retrieve low-level information about current memory usage
- as used by TSynMonitorMemory
- under BSD, only memtotal/memfree/percent are properly returned
- allocreserved and allocused are set only if withalloc is TRUE

function HtmlEscapeMarkdown(const md: RawUTF8; esc: TTextWriterHTMLEscape=[heEmojiToUTF8]): RawUTF8;

Escape some Markdown-marked text into HTML
- just a wrapper around AddHtmlEscapeMarkdown() process

function HtmlEscapeWiki(const wiki: RawUTF8; esc: TTextWriterHTMLEscape=[heHtmlEscape,heEmojiToUTF8]): RawUTF8;

Escape some wiki-marked text into HTML
- just a wrapper around AddHtmlEscapeWiki() process

function InlineParameter(ID: Int64): shortstring; overload;

Returns a 64-bit value as inlined ':1234:' text

function InlineParameter(const value: RawUTF8): RawUTF8; overload;

Returns a string value as inlined ':value:' text

function IPToCardinal(const aIP: RawUTF8): cardinal; overload;

Convert an IPv4 'x.x.x.x' text into its 32-bit value, 0 or localhost
- returns <> 0 value if the text was a valid IPv4 text, 0 on parsing error
- " or '127.0.0.1' will also return 0

function IPToCardinal(P: PUTF8Char; out aValue: cardinal): boolean; overload;

Convert an IPv4 'x.x.x.x' text into its 32-bit value
- returns TRUE if the text was a valid IPv4 text, unserialized as 32-bit aValue
- returns FALSE on parsing error, also setting aValue=0
- " or '127.0.0.1' will also return false
function IPToCardinal(const aIP: RawUTF8; var aValue: cardinal): boolean; overload;

Convert an IPv4 'x.x.x.x' text into its 32-bit value
- returns TRUE if the text was a valid IPv4 text, unserialized as 32-bit aValue
- returns FALSE on parsing error, also setting aValue=0
- "' or '127.0.0.1' will also return false

function IsEqual(const A,B: TSQLFieldBits): boolean; overload;

Fast comparison of two TSQLFieldBits values
- is optimized for 64, 128, 192 and 256 max bits count (i.e. MAX_SQLFIELDS)
- will work also with any other value

function IsMatch(const Pattern, Text: RawUTF8; CaseInsensitive: boolean=false): boolean;

Return TRUE if the supplied content matches a glob pattern
- ?  Matches any single character
- *  Matches any contiguous characters
- [abc]  Matches a or b or c at that position
- [^abc]  Matches anything but a or b or c at that position
- [labc]  Matches anything but a or b or c at that position
- [a-e]  Matches a through e at that position
- [abc-z]  Matches a or b or c or x or y or z, as does [a-cx-z]
- [ma?ch.]* would match match.exe, mavch.dat, march.on, etc..
- 'this [e-n]s a [iz]est' would match 'this is a test', but would not match 'this as a test' nor 'this is a zest'
- consider using TMatch or TMatches if you expect to reuse the pattern

function IsMatchString(const Pattern, Text: string; CaseInsensitive: boolean=false): boolean;

Return TRUE if the supplied content matches a glob pattern, using VCL strings
- is a wrapper around IsMatch() with fast UTF-8 conversion

function Iso8601ToSQL(const S: RawByteString): RawUTF8;

Convert a Iso8601 encoded string into a ISO-8601 string format for SQL '? inlined parameters
- follows the same pattern as DateToSQL or DateTimeToSQL functions, i.e. will return the date or time encoded as \uff11YYYY-MM-DDThh:mm:ss' - therefore ':("\uff11201205-04T20:12:13")' pattern will be recognized as a sftDateTime inline parameter in SQLParamContent() / ExtractsInlineParameters() (JSON_SQLDATE_MAGIC will be used as prefix to create \uff11... pattern)
- in practice, just append the JSON_SQLDATE_MAGIC prefix to the supplied text

function IsValidEmail(P: PUTF8Char): boolean;

Return TRUE if the supplied content is a valid email address
- follows RFC 822, to validate local-part@domain email format

function IsValidIP4Address(P: PUTF8Char): boolean;

Return TRUE if the supplied content is a valid IP v4 address

function IsZero(const Fields: TSQLFieldBits): boolean; overload;

Returns TRUE if no bit inside this TSQLFieldBits is set
- is optimized for 64, 128, 192 and 256 max bits count (i.e. MAX_SQLFIELDS)
- will work also with any other value
function MatchAdd(const One: TMatch; var Several: TMatchDynArray): boolean;
Add one TMatch if not already registered in the Several[] dynamic array

function MatchAny(const Match: TMatchDynArray; const Text: RawUTF8): boolean;
Returns TRUE if Match=nil or if any Match[].Match(Text) is TRUE

function MatchExists(const One: TMatch; const Several: TMatchDynArray): boolean;
Search if one TMatch is already registered in the Several[] dynamic array

function NullableBoolean(Value: boolean): TNullableBoolean;
Creates a nullable Boolean value from a supplied constant
- FPC does not allow direct assignment to a TNullableBoolean = type variant variable: use this function to circumvent it

function NullableBooleanIsEmptyOrNull(const V: TNullableBoolean): Boolean;
Same as VarIsEmpty(V) or VarIsEmpty(V), but faster
- FPC VarIsNull() seems buggy with varByRef variants, and does not allow direct transtyping from a TNullableBoolean = type variant variable: use this function to circumvent those limitations

function NullableBooleanToValue(const V: TNullableBoolean; out Value: Boolean): Boolean; overload;
Check if a TNullableBoolean is null, or return its value
- returns FALSE if V is null or empty, or TRUE and set the Boolean value

function NullableBooleanToValue(const V: TNullableBoolean): Boolean; overload;
Check if a TNullableBoolean is null, or return its value
- returns false if V is null or empty, or the stored Boolean value

function NullableCurrency(const Value: currency): TNullableCurrency;
Creates a nullable Currency value from a supplied constant
- FPC does not allow direct assignment to a TNullableCurrency = type variant variable: use this function to circumvent it

function NullableCurrencyIsEmptyOrNull(const V: TNullableCurrency): Boolean;
Same as VarIsEmpty(V) or VarIsEmpty(V), but faster
- FPC VarIsNull() seems buggy with varByRef variants, and does not allow direct transtyping from a TNullableCurrency = type variant variable: use this function to circumvent those limitations

function NullableCurrencyToValue(const V: TNullableCurrency): currency; overload;
Check if a TNullableCurrency is null, or return its value
- returns 0 if V is null or empty, or the stored Currency value

function NullableCurrencyToValue(const V: TNullableCurrency; out Value: currency): boolean; overload;
Check if a TNullableCurrency is null, or return its value
- returns FALSE if V is null or empty, or TRUE and set the Currency value

function NullableDateTime(const Value: TDateTime): TNullableDateTime;
Creates a nullable TDateTime value from a supplied constant
- FPC does not allow direct assignment to a TNullableDateTime = type variant variable: use this function to circumvent it
function NullableDateTimeIsEmptyOrNull(const V: TNullableDateTime): Boolean;

Same as VarIsEmpty(V) or VarIsEmpty(V), but faster
- FPC VarIsNull() seems buggy with varByRef variants, and does not allow direct transtyping from a TNullableDateTime = type variant variable: use this function to circumvent those limitations

function NullableDateTimeToValue(const V: TNullableDateTime; out Value: TDateTime): boolean; overload;

- Check if a TNullableDateTime is null, or return its value
  - returns FALSE if V is null or empty, or TRUE and set the DateTime value

function NullableDateTimeToValue(const V: TNullableDateTime): TDateTime; overload;

- Check if a TNullableDateTime is null, or return its value
  - returns 0 if V is null or empty, or the stored DateTime value

function NullableFloat(const Value: double): TNullableFloat;

Creates a nullable floating-point value from a supplied constant
- FPC does not allow direct assignment to a TNullableFloat = type variant variable: use this function to circumvent it

function NullableFloatIsEmptyOrNull(const V: TNullableFloat): Boolean;

Same as VarIsEmpty(V) or VarIsEmpty(V), but faster
- FPC VarIsNull() seems buggy with varByRef variants, and does not allow direct transtyping from a TNullableFloat = type variant variable: use this function to circumvent those limitations

function NullableFloatToValue(const V: TNullableFloat): double; overload;

- Check if a TNullableFloat is null, or return its value
  - returns 0 if V is null or empty, or the stored Float value

function NullableFloatToValue(const V: TNullableFloat; out Value: double): boolean; overload;

- Check if a TNullableFloat is null, or return its value
  - returns FALSE if V is null or empty, or TRUE and set the Float value

function NullableInteger(const Value: Int64): TNullableInteger;

Creates a nullable integer value from a supplied constant
- FPC does not allow direct assignment to a TNullableInteger = type variant variable: use this function to circumvent it

function NullableIntegerIsEmptyOrNull(const V: TNullableInteger): Boolean;

Same as VarIsEmpty(V) or VarIsEmpty(V), but faster
- FPC VarIsNull() seems buggy with varByRef variants, and does not allow direct transtyping from a TNullableInteger = type variant variable: use this function to circumvent those limitations

function NullableIntegerToValue(const V: TNullableInteger; out Value: Int64): Boolean; overload;

- Check if a TNullableInteger is null, or return its value
  - returns FALSE if V is null or empty, or TRUE and set the Integer value

function NullableIntegerToValue(const V: TNullableInteger): Int64; overload;

- Check if a TNullableInteger is null, or return its value
  - returns 0 if V is null or empty, or the stored Integer value
function NullableTimeLog(const Value: TTimeLog): TNullableTimeLog;
Creates a nullable TTimeLog value from a supplied constant
- FPC does not allow direct assignment to a TNullableTimeLog = type variant variable: use this
function to circumvent it

function NullableTimeLogIsEmptyOrNull(const V: TNullableTimeLog): Boolean;
Same as VarIsEmpty(V) or VarIsEmpty(V), but faster
- FPC VarIsNil() seems buggy with varByRef variants, and does not allow direct transtyping from a
TNullableTimeLog = type variant variable: use this function to circumvent those limitations

function NullableTimeLogToValue(const V: TNullableTimeLog; out Value: TTimeLog): boolean; overload;
Check if a TNullableTimeLog is null, or return its value
- returns FALSE if V is null or empty, or TRUE and set the TimeLog value

function NullableTimeLogToValue(const V: TNullableTimeLog): TTimeLog; overload;
Check if a TNullableTimeLog is null, or return its value
- returns 0 if V is null or empty, or the stored TimeLog value

function NullableUTF8Text(const Value: RawUTF8): TNullableUTF8Text;
Creates a nullable UTF-8 encoded text value from a supplied constant
- FPC does not allow direct assignment to a TNullableUTF8 = type variant variable: use this
function to circumvent it

function NullableUTF8TextIsEmptyOrNull(const V: TNullableUTF8Text): Boolean;
Same as VarIsEmpty(V) or VarIsEmpty(V), but faster
- FPC VarIsNil() seems buggy with varByRef variants, and does not allow direct transtyping from a
TNullableUTF8Text = type variant variable: use this function to circumvent those limitations

function NullableUTF8TextToValue(const V: TNullableUTF8Text): RawUTF8; overload;
Check if a TNullableUTF8Text is null, or return its value
- returns "" if V is null or empty, or the stored UTF8-encoded text value

function NullableUTF8TextToValue(const V: TNullableUTF8Text; out Value: RawUTF8): boolean; overload;
Check if a TNullableUTF8Text is null, or return its value
- returns FALSE if V is null or empty, or TRUE and set the UTF8Text value

function SearchFieldIndex(var Indexes: TSQLFieldIndexDynArray; Field: integer): integer;
Search a field index in an array of field indexes
- returns the index in Indexes[] of the given Field value, -1 if not found

function SetMatchs(const CSVPattern: RawUTF8; CaseInsensitive: boolean; out Match: TMatchDynArray): integer; overload;
Fill the Match[] dynamic array with all glob patterns supplied as CSV
- returns how many patterns have been set in Match[]
- note that the CSVPattern instance should remain in memory, since it will be pointed to by the
Match[],Pattern private field
function SetMatchs(CSVPattern: PUTF8Char; CaseInsensitive: boolean; Match: PMatch; MatchMax: integer): integer; overload;

*Fill the Match[0..MatchMax] static array with all glob patterns supplied as CSV*
- note that the CSVPattern instance should remain in memory, since it will be pointed to by the Match[].Pattern private field

function SoundExAnsi(A: PAnsiChar; next: PPAnsiChar=nil; Lang: TSynSoundExPronunciation=sndxEnglish): cardinal; overload;

*Retrieve the Soundex value of a text word, from Ansi buffer*
- Return the soundex value as an easy to use cardinal value, 0 if the incoming string contains no valid word
- if next is defined, its value is set to the end of the encoded word (so that you can call again this function to encode a full sentence)

function SoundExAnsi(A: PAnsiChar; next: PPAnsiChar; Lang: PSoundExValues): cardinal;

*Retrieve the Soundex value of a text word, from Ansi buffer*
- Return the soundex value as an easy to use cardinal value, 0 if the incoming string contains no valid word
- if next is defined, its value is set to the end of the encoded word (so that you can call again this function to encode a full sentence)

function SoundExUTF8(U: PUTF8Char; next: PPUTF8Char=nil; Lang: TSynSoundExPronunciation=sndxEnglish): cardinal;

*Retrieve the Soundex value of a text word, from UTF-8 buffer*
- Return the soundex value as an easy to use cardinal value, 0 if the incoming string contains no valid word
- if next is defined, its value is set to the end of the encoded word (so that you can call again this function to encode a full sentence)
- very fast: all UTF-8 decoding is handled on the fly

function SQLParamContent(P: PUTF8Char; out ParamType: TSQLParamType; out ParamValue: RawUTF8; out wasNull: boolean): PUTF8Char;

*Guess the content type of an UTF-8 SQL value, in :(...) format*
- will be used e.g. by ExtractInlineParameters() to un-inline a SQL statement
- sftInteger is returned for an INTEGER value, e.g. :(1234):
- sftFloat is returned for any floating point value (i.e. some digits separated by a '.' character), e.g. :(12.34): or :(12E-34):
- sftUTF8Text is returned for :("text"): or :('text'):, with double quoting inside the value
- sftBlob will be recognized from the ':'\(\uFF00base64encodedbinary\)': pattern, and return raw binary (for direct blob parameter assignment)
- sftDateTime will be recognized from \(\uFF12012-05-04\)' pattern, i.e. JSON_SQLDATE_MAGIC-prefixed string as returned by DateToSQL() or DateTimeToSQL() functions
- sftUnknown is returned on invalid content, or if wasNull is set to TRUE
- if ParamValue is not nil, the pointing RawUTF8 string is set with the value inside ;(...) with double quoting in case of sftUTF8Text
- wasNull is set to TRUE if P was ;'(null)': and ParamType is sftUnknown

function SQLToDateTime(const ParamValueWithMagic: RawUTF8): TDateTime;

*Decode a SQL '?' inlined parameter (i.e. with JSON_SQLDATE_MAGIC prefix)*
- as generated by DateToSQL/DateTimeToSQL/TimeLogToSQL functions
function SQLVarLength(const Value: TSQLVar): integer;
    Returns the stored size of a TSQLVar database value
    - only returns VBlobLen / StrLen(VText) size, 0 otherwise

function StringToConsole(const S: string): RawByteString;
    Direct conversion of a VCL string into a console OEM-encoded String
    - under Windows, will use the CP_OEMCP encoding
    - under Linux, will expect the console to be defined with UTF-8 encoding

function SynTableVariantVarType: cardinal;
    Initialize TSynTableVariantType if needed, and return the corresponding VType

function SystemInfoJson: RawUTF8;
    Returns a JSON object containing basic information about the computer
    - including Host, User, CPU, OS, freemem, freedisk...

procedure TextBackground(Color: TConsoleColor);
    Change the console text background color

procedure TextColor(Color: TConsoleColor);
    Change the console text writing color
    - you should call this procedure to initialize StdOut global variable, if you manually initialized the Windows console, e.g. via the following code:
      AllocConsole;
      TextColor(ccLightGray); // initialize internal console context

function TextToSQLDBFieldType(json: PUTF8Char): TSQLDBFieldType;
    Guess the correct TSQLDBFieldType from the UTF-8 representation of a value

function TimeLogToSQL(const Timestamp: TTimeLog): RawUTF8;
    Convert a TTimeLog value into a ISO-8601 string format for SQL '?' inlined parameters
    - handle TTimeLog bit-encoded Int64 format
    - follows the same pattern as DateToSQL or DateTimeToSQL functions, i.e. will return the date or time encoded as '\uFFF1YYYY-MM-DDThh:mm:ss' - therefore ':("\uFFF12012-05-04T20:12:13");' pattern will be recognized as a sftDateTime inline parameter in SQLParamContent() / ExtractInlineParameters() (JSON_SQLDATE_MAGIC will be used as prefix to create '\uFFF1...' pattern)
    - to be used e.g. as in:
      aRec.CreateAndFillPrepare(Client, 'Datum<=?', [TimeLogToSQL(TimeLogNow)]);

procedure ToSBFStr(const Value: RawByteString; out Result: TSBFString);
    Convert any AnsiString content into our SBF compact binary format storage

function Utf8ToConsole(const S: RawUTF8): RawByteString;
    Direct conversion of a UTF-8 encoded string into a console OEM-encoded String
    - under Windows, will use the CP_OEMCP encoding
    - under Linux, will expect the console to be defined with UTF-8 encoding
procedure VariantToSQLVar(const Input: variant; var temp: RawByteString; var Output: TSQLVar);

Convert any Variant into a database value
- ftBlob kind won't be handled by this function
- complex variant types would be converted into ftUTF8 JSON object/array

function VariantToSQLDBFieldType(const V: Variant): TSQLDBFieldType;

Guess the correct TSQLDBFieldType from a variant value

function VariantVTypeToSQLDBFieldType(VType: cardinal): TSQLDBFieldType;

Guess the correct TSQLDBFieldType from a variant type

procedure ZeroCompress(P: PAnsiChar; Len: integer; Dest: TFileBufferWriter);

RLE compression of a memory buffer containing mostly zeros
- will store the number of consecutive zeros instead of plain zero bytes
- used for spare bit sets, e.g. TSynBloomFilter serialization
- will also compute the crc32c of the supplied content
- use ZeroDecompress() to expand the compressed result
- resulting content would be at most 14 bytes bigger than the input
- you may use this function before SynLZ compression

procedure ZeroCompressXor(New,Old: PAnsiChar; Len: cardinal; Dest: TFileBufferWriter);

RLE compression of XORed memory buffers resulting in mostly zeros
- will perform ZeroCompress(Dest^ := New^ xor Old^) without any temporary memory allocation
- is used e.g. by TSynBloomFilterDiff.SaveToDiff() in incremental mode
- will also compute the crc32c of the supplied content

procedure ZeroDecompress(P: PByte; Len: integer; out Dest: RawByteString);

RLE uncompression of a memory buffer containing mostly zeros
- returns Dest='' if P^ is not a valid ZeroCompress() function result
- used for spare bit sets, e.g. TSynBloomFilter serialization
- will also check the crc32c of the supplied content

function ZeroDecompressOr(P,Dest: PAnsiChar; Len,DestLen: integer): boolean;

RLE uncompression and ORing of a memory buffer containing mostly zeros
- will perform Dest^ := Dest^ or ZeroDecompress(P^) without any temporary memory allocation
- is used e.g. by TSynBloomFilterDiff.LoadFromDiff() in incremental mode
- returns false if P^ is not a valid ZeroCompress/ZeroCompressXor() result
- will also check the crc32c of the supplied content

Variables implemented in the SynTable unit

EMOJI_AFTERDOTS: array["' '..' "] of TEmoji;
  To recognize simple :) :( :| :/ :D :o :p :s characters as smileys

EMOJI_RTTI: PShortString;
  Low-level access to TEmoji RTTI - used when inlining EmojiFromText()

EMOJI_TAG: array[TEmoji] of RawUTF8;
  Github/Markdown compatible tag of Emojis, including trailing and ending :
  - e.g. ‘grinning:’ or ‘:person_with_pouting_face:’
EMOJI_TEXT: array[TEmoji] of RawUTF8;

GitHub/Markdown compatible text of Emojis
- e.g. 'grinning' or 'person_with_pouting_face'

EMOJI_UTF8: array[TEmoji] of RawUTF8;
The Unicode character matching a given Emoji, after UTF-8 encoding

NullableBooleanNull: TNullableBoolean absolute NullVarData;
A nullable boolean value containing null

NullableCurrencyNull: TNullableCurrency absolute NullVarData;
A nullable currency value containing null

NullableDateTimeNull: TNullableDateTime absolute NullVarData;
A nullable TDateTime value containing null

NullableFloatNull: TNullableFloat absolute NullVarData;
A nullable float value containing null

NullableIntegerNull: TNullableInteger absolute NullVarData;
A nullable integer value containing null

NullableTimeLogNull: TNullableTimeLog absolute NullVarData;
A nullable TTimeLog value containing null

NullableUTF8TextNull: TNullableUTF8Text absolute NullVarData;
A nullable UTF-8 encoded text value containing null

StdOut: THandle;
Low-level handle used for console writing
- may be overridden when console is redirected
- is initialized when TextColor() is called

TSynPersistentWithPasswordUserCrypt: function(const Data,AppServer: RawByteString; Encrypt: boolean): RawByteString;

Function prototype to customize TSynPersistent class password storage
- is called when 'user1:base64pass1,user2:base64pass2' layout is found, and the current user logged on the system is user1 or user2
- you should not call this low-level method, but assign e.g. from SynCrypto:
  TSynPersistentWithPasswordUserCrypt := CryptDataForCurrentUser;
27.41. SynTaskDialog.pas unit

Purpose: Implement TaskDialog window (native on Vista/Seven, emulated on XP)
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Objects implemented in the SynTaskDialog unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEmulatedTaskDialog</td>
<td>The actual form class used for emulation</td>
<td>1824</td>
</tr>
<tr>
<td>TSynButton</td>
<td>A generic Button to be used in the User Interface</td>
<td>1828</td>
</tr>
<tr>
<td>TTaskDialog</td>
<td>Implements a TaskDialog</td>
<td>1825</td>
</tr>
<tr>
<td>TTaskDialogEx</td>
<td>A wrapper around the TTaskDialog.Execute method</td>
<td>1827</td>
</tr>
<tr>
<td>TTaskDialogImplementation</td>
<td>Structure for low-level access to the task dialog implementation</td>
<td>1825</td>
</tr>
</tbody>
</table>

TEmulatedTaskDialog = class(TForm)

The actual form class used for emulation

Combo: TComboBox;

The Task Dialog selection list

Edit: TEdit;

The Task Dialog optional query editor

Element: array[tdeContent..tdeMainInstruction] of TLabel;

The labels corresponding to the Task Dialog main elements

Owner: PTaskDialog;

The Task Dialog structure which created the form

Verif: TCheckBox;

The Task Dialog optional checkbox
TTaskDialogImplementation = record

Structure for low-level access to the task dialog implementation
- points either to the HWND handle of the new TaskDialog API or to the emulation dialog

TTaskDialog = object(TObject)

Implements a TaskDialog
- will use the new TaskDialog API under Vista/Seven, and emulate it with pure Delphi code and standard themed VCL components under XP or 2K
- create a TTaskDialog object/record on the stack will initialize all its string parameters to " (it's a SHAME that since Delphi 2009, objects are not initialized any more: we have to define this type as object before Delphi 2009, and as record starting with Delphi 2009)
- set the appropriate string parameters, then call Execute() with all additional parameters
- RadioRes/SelectionRes/VerifyChecked will be used to reflect the state after dialog execution
- here is a typical usage:

var
Task: TTaskDialog;

begin
Task.Inst := 'Saving application settings';
Task.Content := 'This is the content';
Task.Radios := 'Store settings in registry' #10 'Store settings in XML file';
Task.Verify := 'Do no ask for this setting next time';
Task.VerifyChecked := true;
Task.Footer := 'XML file is perhaps a better choice';
Task.Execute([], 0, [], tiQuestion, tfiInformation, mrRad1);
ShowMessage(IntToStr(Task.RadioRes)); // mrRad1=Registry, mrRad2=XML
if Task.VerifyChecked then
ShowMessage(Task.Verify);
end;

Buttons: string;

A #13#10 or #10 separated list of custom buttons
- they will be identified with an ID number starting at 100 (so you may use mrBtn1, mrBtn2, mrBtn3... mrBtn9 constants)
- by default, the buttons will be created at the dialog bottom, just like the common buttons
- if tdfUseCommandLinks flag is set, the custom buttons will be created as big button in the middle of the dialog window; in this case, any \n will be converted as note text (shown with smaller text under native Vista/Seven TaskDialog, or as popup hint within Delphi emulation)
- see the AddButton() wrapper method for an easy access

Content: string;

The dialog's primary content content text
- any \n will be converted into a line feed

Dialog: TTaskDialogImplementation;

Low-level access to the task dialog implementation

Footer: string;

The footer content text
- any \n will be converted into a line feed
Info: string;
The expanded information content text
- any \n will be converted into a line feed
- the Delphi emulation will always show the Info content (there is no collapse/expand button)

InfoCollapse: string;
The button caption to be displayed when the information is expanded
- not used under XP: the Delphi emulation will always show the Info content

InfoExpanded: string;
The button caption to be displayed when the information is collapsed
- not used under XP: the Delphi emulation will always show the Info content

Inst: string;
The main instruction (first line on top of window)
- any \n will be converted into a line feed
- if left void, the text is taken from the current dialog icon kind

Query: string;
Some text to be edited
- if tdfQuery is in the flags, will contain the default query text
- if Selection is set, the

RadioRes: integer;
The selected radio item
- first is numeroted 0

Radios: string;
A #13#10 or #10 separated list of custom radio buttons
- they will be identified with an ID number starting at 200 (so you may use mrRad1, mrRad2, mrRad3... mrRad9 constants)
- aRadioDef parameter can be set to define the default selected value
- \n will be converted as note text (shown with smaller text under native Vista/Seven TaskDialog, or as popup hint within Delphi emulation)

Selection: string;
A #13#10 or #10 separated list of items to be selected
- if set, a Combo Box will be displayed to select
- if tdfQuery is in the flags, the combo box will be in edition mode, and the user will be able to edit the Query text or fill the field with one item of the selection
- this selection is not handled via the Vista/Seven TaskDialog, but with our Delphi emulation code (via a TComboBox)

SelectionRes: integer;
After execution, contains the selected item from the Selection list

Title: string;
The main title of the dialog window
- if left void, the title of the application main form is used

Verify: string;
The text of the bottom most optional checkbox
VerifyChecked: BOOL;

Reflect the the bottom most optional checkbox state
- if Verify is not '', should be set before execution
- after execution, will contain the final checkbox state

function Execute(aCommonButtons: TCommonButtons=[]; aButtonDef: integer=0; aFlags: TTaskDialogFlags=[]; aDialogIcon: TTaskDialogIcon=tiInformation; aFooterIcon: TTaskDialogFooterIcon=tfiWarning; aRadioDef: integer=0; aWidth: integer=0; aParent: HWND=0; aNonNative: boolean=false; aEmulateClassicStyle: boolean = false; aOnButtonClicked: TTaskDialogButtonClickedEvent=nil): integer;

Launch the TaskDialog form
- some common buttons can be set via aCommonButtons
- in emulation mode, aFlags will handle only tdfUseCommandLinks, tdfUseCommandLinksNoIcon, and tdfQuery options
- will return 0 on error, or the Button ID (e.g. mrOk for the OK button or mrBtn1/100 for the first custom button defined in Buttons string)
- if Buttons was defined, aButtonDef can set the selected Button ID
- if Radios was defined, aRadioDef can set the selected Radio ID
- aDialogIcon and aFooterIcon are used to specify the displayed icons
- aWidth can be used to force a custom form width (in pixels)
- aParent can be set to any HWND - by default, Application.DialogHandle
- if aNonNative is TRUE, the Delphi emulation code will always be used
- aEmulateClassicStyle can be set to enforce conformity with the non themed user interface - see @https://synopse.info/forum/viewtopic.php?pid=2867#p2867
- aOnButtonClicked can be set to a callback triggered when a button is clicked

procedure AddButton(const ACaption: string; const ACommandLinkHint: string = '');
Wrapper method able to add a custom button to the Task Dialog
- will add the expected content to the Buttons text field

procedure SetElementText(element: TTaskDialogElement; const Text: string);
Allow a OnButtonClicked callback to change the Task Dialog main elements
- note that tdeVerif could be modified only in emulation mode, since the API does not give any runtime access to the checkbox caption
- other elements will work in both emulated and native modes

TTTaskDialogEx = object(TObject)
A wrapper around the TTaskDialog.Execute method
- used to provide a "flat" access to task dialog parameters

Base: TTaskDialog;
The associated main TTaskDialog instance

ButtonDef: integer;
The default button ID

CommonButtons: TCommonButtons;
Some common buttons to be displayed

DialogIcon: TTaskDialogIcon;
Used to specify the dialog icon
EmulateClassicStyle: boolean;
   *Can be used to enforce conformity with the non themed user interface*

Flags: TTaskDialogFlags;
   *The associated configuration flags for this Task Dialog*
   - in emulation mode, aFlags will handle only tdfUseCommandLinks,
     tdfUseCommandLinksNoIcon, and tdfQuery options

FooterIcon: TTaskDialogFooterIcon;
   *Used to specify the footer icon*

NonNative: boolean;
   *If TRUE, the Delphi emulation code will always be used*

OnButtonClicked: TTaskDialogButtonClickedEvent;
   *This event handler will be fired on a button dialog click*

RadioDef: integer;
   *The default radio button ID*

Width: integer;
   *Can be used to force a custom form width (in pixels)*

**function** Execute(aParent: HWND=0): integer;
   *Main (and unique) method showing the dialog itself*
   - is in fact a wrapper around the TTaskDialog.Execute method

**procedure** Init;
   *Will initialize the dialog parameters*
   - can be used to display some information with less parameters:

```pascal
var
   TaskEx: TTaskDialogEx;
   ...
   TaskEx.Init;
   TaskEx.Base.Title := 'Task Dialog Test';
   TaskEx.Base.Inst := 'Callback Test';
   TaskEx.Execute;
```

---

**TSynButton = class(TSynButtonParent)**

*A generic Button to be used in the User Interface*
   - is always a Themed button: under Delphi 6, since TBitBtn is not themed, it will be a row TButton
     with no glyph... never mind...

**constructor** CreateKind(Owner: TWinControl; Btn: TCommonButton; Left, Right, Width, Height: integer);
   *
     Create a standard button instance
   *
   - ModalResult/Default/Cancel properties will be set as expected for this kind of button

**procedure** DoDropDown;
   *Drop down the associated Popup Menu*

**procedure** SetBitmap(Bmp: TBitmap);
   *Set the glyph of the button*
   - set nothing under Delphi 6
property DropDownMenu: TSynPopupMenu read fDropDownMenu write fDropDownMenu;

The associated Popup Menu to drop down

Types implemented in the SynTaskDialog unit

TCommonButton = ( cbOK, cbYes, cbNo, cbCancel, cbRetry, cbClose );

The standard kind of common buttons handled by the Task Dialog

TCommonButtons = set of TCommonButton;

Set of standard kind of common buttons handled by the Task Dialog

TSynPopupMenu = TPopupMenu;

A generic VCL popup menu

TTaskDialogButtonClickedEvent = procedure(Sender: PTaskDialog; AButtonID: integer; var ACanClose: Boolean) of object;

This callback will be triggered when a task dialog button is clicked
- to prevent the task dialog from closing, the application must set ACanClose to FALSE, otherwise the task dialog is closed and the button ID is returned via the original TTaskDialog.Execute() result

TTaskDialogElement = ( tdeContent, tdeExpandedInfo, tdeFooter, tdeMainInstruction, tdeEdit, tdeVerif );

The visual components of this Task Dialog
- map low-level TDE_CONTENT...TDE_MAIN_INSTRUCTION constants and the query editor and checkbox
- tdeEdit is for the query editor
- tdeVerif is for the checkbox

TTTaskDialogFlag = ( tdfEnableHyperLinks, tdfUseHIconMain, tdfUseHIconFooter, tdfAllowDialogCancellation, tdfUseCommandLinks, tdfUseCommandLinksNoIcon, tdfExpandFooterArea, tdfExpandByDefault, tdfVerificationFlagChecked, tdfShowProgressBar, tdfShowMarqueeProgressBar, tdfCallbackTimer, tdfPositionRelativeToWindow, tdfRtlLayout, tdfCanBeMinimized, tdfQuery, tdfQueryMasked, tdfQueryFieldFocused );

The available configuration flags for the Task Dialog
- most are standard TDF_* flags used for Vista/Seven native API (see http://msdn.microsoft.com/en-us/library/bb787473(v=vs.85).aspx for TASKDIALOG_FLAGS)
- tdfQuery and tdfQueryMasked are custom flags, implemented in pure Delphi code to handle input query
- our emulation code will handle only tdfUseCommandLinks, tdfUseCommandLinksNoIcon, and tdfQuery options

TTTaskDialogFlags = set of TTaskDialogFlag;

Set of available configuration flags for the Task Dialog

TTaskDialogFooterIcon = ( tfiBlank, tfiWarning, tfiQuestion, tfiError, tfiInformation, tfiShield );

The available footer icons for the Task Dialog

TTTaskDialogIcon = ( tiBlank, tiWarning, tiQuestion, tiError, tiInformation, tiNotUsed, tiShield );

The available main icons for the Task Dialog

Constants implemented in the SynTaskDialog unit
mrBtn1 = 100;
    Match the 1st custom button ID

mrBtn2 = 101;
    Match the 2nd custom button ID

mrBtn3 = 102;
    Match the 3rd custom button ID

mrBtn4 = 103;
    Match the 4th custom button ID

mrBtn5 = 104;
    Match the 5th custom button ID

mrBtn6 = 105;
    Match the 6th custom button ID

mrBtn7 = 106;
    Match the 7th custom button ID

mrBtn8 = 107;
    Match the 8th custom button ID

mrBtn9 = 108;
    Match the 9th custom button ID

mrRad1 = 200;
    Match the 1st custom radio ID

mrRad2 = 201;
    Match the 2nd custom radio ID

mrRad3 = 202;
    Match the 3rd custom radio ID

mrRad4 = 203;
    Match the 4th custom radio ID

mrRad5 = 204;
    Match the 5th custom radio ID

mrRad6 = 205;
    Match the 6th custom radio ID

mrRad7 = 206;
    Match the 7th custom radio ID

mrRad8 = 207;
    Match the 8th custom radio ID

mrRad9 = 208;
    Match the 9th custom radio ID

Functions or procedures implemented in the *SynTaskDialog* unit
Functions or procedures | Description | Page
---|---|---
UnAmp | Return the text without the ' & ' characters within | 1831

```pascal
function UnAmp(const s: string): string;
    Return the text without the ' & ' characters within
```

Variables implemented in the `SynTaskDialog` unit

- **BitmapArrow**: TBitmap;
  - *Will map a generic Arrow picture from SynTaskDialog.res*

- **BitmapOK**: TBitmap;
  - *Will map a generic OK picture from SynTaskDialog.res*

- **DefaultFont**: TFont;
  - *Will map a default font, according to the available*
  - - if Calibri is installed, will use it
  - - will fall back to Tahoma otherwise

- **DefaultTaskDialog**: TTaskDialogEx = (DialogIcon: tiInformation; FooterIcon: tfiWarning);
  - *A default Task Dialog wrapper instance*
  - - can be used to display some information with less parameters, just like the TTaskDialogEx.Init method:
  - ```pascal
  var TaskEx: TTaskDialogEx;
  ... TaskEx := DefaultTaskDialog;
  TaskEx.Base.Title := 'Task Dialog Test';
  TaskEx.Base.Inst := 'Callback Test';
  TaskEx.Execute;
  ```

- **TaskDialogIndirect**: function(AConfig: pointer; Res: PInteger; ResRadio: PInteger; VerifyFlag: PBOOL): HRESULT;
  - *Is filled once in the initialization block below*
  - - you can set this reference to nil to force Delphi dialogs even on Vista/Seven (e.g. make sense if TaskDialogBiggerButtons=true)
27.42. SynTests.pas unit

**Purpose:** Unit test functions used by Synopse projects
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynTests unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
</tbody>
</table>

**Objects implemented in the SynTests unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSynTest</td>
<td>Abstract parent class for both tests suit (TSynTests) and cases</td>
<td>1833</td>
</tr>
<tr>
<td>TSynTestCase</td>
<td>A class implementing a test case</td>
<td>1834</td>
</tr>
<tr>
<td>TSynTestFailed</td>
<td>Information about a failed test</td>
<td>1837</td>
</tr>
<tr>
<td>TSynTestMethodInfo</td>
<td>How published method information is stored within TSynTest</td>
<td>1832</td>
</tr>
<tr>
<td>TSynTests</td>
<td>A class used to run a suit of test cases</td>
<td>1837</td>
</tr>
<tr>
<td>TSynTestsLogged</td>
<td>This overridden class will create a .log file in case of a test case failure</td>
<td>1839</td>
</tr>
</tbody>
</table>

**TSynTestMethodInfo** = record

*How published method information is stored within TSynTest*

IdentTestName: RawUTF8;

*Ready-to-be-displayed 'Ident - TestName' text, as UTF-8*
Method: TSynTestEvent;
   Direct access to the method execution

MethodIndex: integer;
   The index of this method in the TSynTestCase

MethodName: RawUTF8;
   Raw method name, as defined in pascal code (not uncamelcased)

Test: TSynTest;
   The test case holding this method

TestName: string;
   The uncamelcased method name

TSynTest = class(TSynPersistent)
Abstract parent class for both tests suit (TSynTests) and cases (TSynTestCase)
- purpose of this ancestor is to have RTTI for its published methods, and to handle a class text
  identifier, or uncamelcase its class name if no identifier was defined
- sample code about how to use this test framework is available in the "Sample\07 - SynTest"
  folder

constructor Create(const Ident: string = ''); reintroduce; virtual;
   Create the test instance
   - if an identifier is not supplied, the class name is used, after T[Syn][Test] left trim and
     un-camel-case
   - this constructor will add all published methods to the internal test list, accessible via the
     Count/TestName/TestMethod properties

procedure Add(const aMethod: TSynTestEvent; const aMethodName: RawUTF8; const
  aIdent: string);
   Register a specified test to this class instance
   - Create will register all published methods of this class, but your code may initialize its own set
     of methods on need

property Count: Integer read GetCount;
   Return the number of tests associated with this class
   - i.e. the number of registered tests by the Register() method PLUS the number of published
     methods defined within this class

property Ident: string read GetIdent;
   The test name
   - either the Ident parameter supplied to the Create() method, either a uncameled text from the
     class name

property InternalTestsCount: integer read fInternalTestsCount;
   Return the number of published methods defined within this class as tests
   - i.e. the number of tests added by the Create() constructor from RTTI
   - any TestName/TestMethod[] index higher or equal to this value has been added by a specific
     call to the Add() method
property Options: TSynTestOptions read fOptions write fOptions;
  Allows to tune the test case process

TSynTestCase = class(TSynTest)
  A class implementing a test case
  - should handle a test unit, i.e. one or more tests
  - individual tests are written in the published methods of this class

  constructor Create(Owner: TSynTests; const Ident: string = ''); reintroduce;
  virtual;
  Create the test case instance
  - must supply a test suit owner
  - if an identifier is not supplied, the class name is used, after T[Syn][Test] left trim and
  un-camel-case

  destructor Destroy; override;
  Clean up the instance
  - will call CleanUp, even if already done before

function CheckEqual(a,b: pointer; const msg: RawUTF8 = ''): Boolean; overload;
  Used by the published methods to run test assertion against pointers/classes
  - if a<>b, will fail and include '#<>#' text before the supplied msg

function CheckEqual(a,b: Int64; const msg: RawUTF8 = ''): Boolean; overload;
  Used by the published methods to run test assertion against integers
  - if a<>b, will fail and include '#<>#' text before the supplied msg

function CheckEqual(const a,b: RawUTF8; const msg: RawUTF8 = ''): Boolean; overload;
  Used by the published methods to run test assertion against UTF-8 strings
  - if a<>b, will fail and include '#<>#' text before the supplied msg

function CheckFailed(condition: Boolean; const msg: string = ''): Boolean;
  Used by the published methods to run a test assertion
  - condition must equals TRUE to pass the test
  - function return TRUE if the condition failed, in order to allow the caller to stop testing with
    such code:
    if CheckFailed(A=10) then exit;

function CheckMatchAny(const Value: RawUTF8; const Values: array of RawUTF8;
CaseSensitive: Boolean=true; ExpectedResult: Boolean=true; const msg: string = ''): Boolean;
  Perform a string comparison with several value
  - test passes if (Value=Values[0]) or (Value=Value[1]) or (Value=Values[2])... and
  ExpectedResult=true
function CheckNot(condition: Boolean; const msg: string = ''): Boolean;

*Used by the published methods to run a test assertion*
- condition must equals FALSE to pass the test
- function return TRUE if the condition failed, in order to allow the caller to stop testing with such code:
  ```pascal
  if CheckNot(A<>10) then exit;
  ```

function CheckNotEqual(a,b: pointer; const msg: RawUTF8 = ''): Boolean; overload;

*Used by the published methods to run test assertion against pointers/classes*
- if a=b, will fail and include '#=#' text before the supplied msg

function CheckNotEqual(const a,b: RawUTF8; const msg: RawUTF8 = ''): Boolean; overload;

*Used by the published methods to run test assertion against UTF-8 strings*
- if a=b, will fail and include '#=#' text before the supplied msg

function CheckNotEqual(a,b: Int64; const msg: RawUTF8 = ''): Boolean; overload;

*Used by the published methods to run test assertion against integers*
- if a=b, will fail and include '#=#' text before the supplied msg

function CheckSame(const Value1,Value2: double; const Precision: double=DOUBLE_SAME; const msg: string = ''): Boolean;

*Used by the published methods to run a test assertion about two double values*
- includes some optional precision argument

function NotifyTestSpeed(const ItemNameFmt: RawUTF8; const ItemNameArgs: array of const;ItemCount: integer; SizeInBytes: cardinal=0; Timer: PPrecisionTimer=nil; OnlyLog: boolean=false): TSynMonitorOneMicroSec; overload;

*Will add to the console a formatted message with a speed estimation*

function NotifyTestSpeed(const ItemName: string; ItemCount: integer; SizeInBytes: cardinal=0; Timer: PPrecisionTimer=nil; OnlyLog: boolean=false): TSynMonitorOneMicroSec; overload;

*Will add to the console a message with a speed estimation*
- speed is computed from the method start
- returns the number of microsec of the (may be specified) timer
- OnlyLog will compute and append the info to the log, but not on the console
- warning: this method is not thread-safe if a local Timer is not specified

class function RandomAnsi7(CharCount: Integer): RawByteString;

*Create a temporary string random content, using ASCII 7 bit content*

class function RandomIdentifier(CharCount: Integer): RawByteString;

*Create a temporary string random content, using A..Z,_,0..9 chars only*

class function RandomString(CharCount: Integer): RawByteString;

*Create a temporary string random content, WinAnsi (code page 1252) content*

class function RandomTextParagraph(WordCount: Integer; LastPunctuation: AnsiChar='.'; const RandomInclude: RawUTF8=''): RawUTF8;

*Create a temporary string, containing some fake text, with paragraphs*
class function RandomUnicode(CharCount: Integer): SynUnicode;
    Create a temporary UTF-16 string random content, using WinAnsi (code page 1252) content

class function RandomURI(CharCount: Integer): RawByteString;
    Create a temporary string random content, using uri-compatible chars only

class function RandomUTF8(CharCount: Integer): RawUTF8;
    Create a temporary UTF-8 string random content, using WinAnsi (code page 1252) content

procedure AddConsole(const msg: string; OnlyLog: boolean=false);
    Append some text to the current console
    - OnlyLog will compute and append the info to the log, but not on the console

class procedure AddRandomTextParagraph(WR: TTextWriter; WordCount: Integer;
    LastPunctuation: AnsiChar='.'; const RandomInclude: RawUTF8=''; NoLineFeed: boolean=false);
    Add containing some "bla bli blo blu" fake text, with paragraphs

procedure Check(condition: Boolean; const msg: string = '');
    Used by the published methods to run a test assertion
    - condition must equals TRUE to pass the test

procedure CheckLogTime(condition: boolean; const msg: RawUTF8; const args: array of const;
    level: TSynLogInfo=sllTrace);
    Used by published methods to write some timing on associated log
    - at least one CheckLogTimeStart method call should happen to reset the internal timer
    - condition must equals TRUE to pass the test
    - the supplied message would be appended, with its timing
    - warning: this method is not thread-safe

procedure CheckLogTimeStart;
    Used by published methods to start some timing on associated log
    - call this once, before one or several consecutive CheckLogTime()
    - warning: this method is not thread-safe

procedure CheckUTF8(condition: Boolean; const msg: RawUTF8); overload;
    Used by the published methods to run a test assertion, with an UTF-8 error message
    - condition must equals TRUE to pass the test

procedure CheckUTF8(condition: Boolean; const msg: RawUTF8; const args: array of const); overload;
    Used by the published methods to run a test assertion, with a error message computed via FormatUTF8()
    - condition must equals TRUE to pass the test

procedure TestFailed(const msg: string);
    This method is triggered internaly - e.g. by Check() - when a test failed

property Assertions: integer read fAssertions;
    The number of assertions (i.e. Check() method call) for this test case

property AssertionsFailed: integer read fAssertionsFailed;
    The number of assertions (i.e. Check() method call) for this test case
property Ident: string read GetIdent;
   The test name
   - either the Ident parameter supplied to the Create() method, either an uncamelied text from the class name

property Owner: TSynTests read fOwner;
   The test suit which owns this test case

TSynTestFailed = record
   Information about a failed test
   Error: string;
      The contextual message associated with this failed test
   IdentTestName: RawUTF8;
      Ready-to-be-displayed 'TestCaseIdent - TestName' text, as UTF-8
   TestName: string;
      The uncamelcased method name

TSynTests = class(TSynTest)
   A class used to run a suit of test cases
   CustomVersions: string;
      You can put here some text to be displayed at the end of the messages
      - some internal versions, e.g.
      - every line of text must explicitly BEGIN with #13#10
   RunTimer: TPrecisionTimer;
      Contains the run elapsed time
   TestTimer: TPrecisionTimer;
      Contains the run elapsed time
   TotalTimer: TPrecisionTimer;
      Contains the run elapsed time

constructor Create(const Ident: string = ''); override;
   Create the test suit
   - if an identifier is not supplied, the class name is used, after T[Syn][Test] left trim and un-camel-case
   - this constructor will add all published methods to the internal test list, accessible via the Count/TestName/TestMethod properties

destructor Destroy; override;
   Finalize the class instance
   - release all registered Test case instance
function Run: Boolean; virtual;

Call of this method will run all associated tests cases
- function will return TRUE if all test passed
- all failed test cases will be added to the Failed[] list - which is cleared at the beginning of the run
- Assertions and AssertionsFailed counter properties are reset and computed during the run
- you may override this method to provide additional information, e.g.

function TMySynTests.Run: Boolean;
begin // need SynSQLite3.pas unit in the uses clause
  CustomVersions := format(#13#10%'Using mORMot %s' + #13#10'    %s %s', [OSVersionText, CpuInfoText,
    SYNOPSE_FRAMEWORK_FULLVERSION, sqlite3.ClassName, sqlite3.Version]);
  result := inherited Run;
end;

procedure AddCase(TestCase: TSynTestCaseClass); overload;

Register a specified Test case from its class name
- an instance of the supplied class will be created during Run
- the published methods of the children must call this method in order to add test cases
- example of use (code from a TSynTests published method):
  AddCase(TOneTestCase);

procedure AddCase(const TestCase: array of TSynTestCaseClass); overload;

Register a specified Test case from its class name
- an instance of the supplied classes will be created during Run
- the published methods of the children must call this method in order to add test cases
- example of use (code from a TSynTests published method):
  AddCase([TOneTestCase]);

class procedure RunAsConsole(const CustomIdent: string=''; withLogs:
  TSynLogInfos=[sllLastError,sllError,sllException,sllExceptionOS]; options:
  TSynTestOptions=[]); virtual;

You can call this class method to perform all the tests on the Console
- it will create an instance of the corresponding class, with the optional identifier to be supplied to its constructor
- if the executable was launched with a parameter, it will be used as file name for the output - otherwise, tests information will be written to the console
- it will optionally enable full logging during the process
- a typical use will first assign the same log class for the whole framework, if the mORMot.pas unit is to be used - in such case, before calling RunAsConsole(), the caller should execute:
  TSynLogTestLog := TSQLLog;
  TMyTestsClass.RunAsConsole('My Automated Tests',LOG_VERBOSE);

procedure SaveToFile(const DestPath: TFileName; const FileName: TFileName='');

Save the debug messages into an external file
- if no file name is specified, the current Ident is used
property Assertions: integer read fAssertions;
   The number of assertions (i.e. Check() method call) in all tests
   - this property is set by the Run method above

property AssertionsFailed: integer read fAssertionsFailed;
   The number of assertions (i.e. Check() method call) which failed in all tests
   - this property is set by the Run method above

property CurrentMethodInfo: PSynTestMethodInfo read fCurrentMethodInfo;
   Method information currently running
   - is set by Run and available within TTestCase methods

property Failed[Index: integer]: TSynTestFailed read GetFailed;
   Retrieve the information associated with a failure

property FailedCount: integer read GetFailedCount;
   Number of failed tests after the last call to the Run method

TSynTestsLogged = class(TSynTests)
   This overridden class will create a .log file in case of a test case failure
   - inherits from TSynTestsLogged instead of TSynTests in order to add logging to your test suite (via a dedicated TSynLogTest instance)

constructor Create(const Ident: string = ''); override;
   Create the test instance and initialize associated LogFile instance
   - this will allow logging of all exceptions to the LogFile

destructor Destroy; override;
   Release associated memory

property LogFile: TSynLog read fLogFile;
   The .log file generator created if any test case failed

Types implemented in the SynTests unit

PSynTestMethodInfo = ^TSynTestMethodInfo;
   Pointer access to published method information

TSynTestCaseClass = class of TSynTestCase;
   Class-reference type (metaclass) of a test case

TSynTestEvent = procedure of object;
   The prototype of an individual test
   - to be used with TSynTest descendants

TSynTestOption = ( tcoLogEachCheck );
   Allows to tune TSynTest process
   - tcoLogEachCheck will log as sICustom4 each non void Check() message

TSynTestOptions = set of TSynTestOption;
   Set of options to tune TSynTest process
27.43. SynVirtualDataSet.pas unit

Purpose: DB VCL read-only virtual dataset
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the SynVirtualDataSet unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/cache/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

SynVirtualDataSet class hierarchy

Objects implemented in the SynVirtualDataSet unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDocVariantArrayDataSet</td>
<td>Read-only virtual TDataSet able to access a dynamic array of TDocVariant</td>
<td>1841</td>
</tr>
<tr>
<td>TSynVirtualDataSet</td>
<td>Read-only virtual TDataSet able to access any content</td>
<td>1840</td>
</tr>
</tbody>
</table>

TSynVirtualDataSet = class(TDataSet)

Read-only virtual TDataSet able to access any content

constructor Create(Owner: TComponent); override;

Class function BcdRead(P: PUTF8Char; var aValue; out aValid: Boolean): PUTF8Char; this overridden constructor will compute an unique Name property

function CreateBlobStream(Field: TField; Mode: TBlobStreamMode): TStream; override;

Get BLOB column data for the current active row
- handle ftBlob, ftMemo, ftWideMemo via GetRowFieldData()

function GetBlobStream(Field: TField; RowIndex: integer): TStream;

Get BLOB column data for a given row (may not the active row)
- handle ftBlob, ftMemo, ftWideMemo via GetRowFieldData()

function GetFieldData(Field: TField; Buffer: pointer): Boolean; override;

Get column data for the current active row
- handle ftBoolean, ftInteger, ftLargeint, ftFloat, ftCurrency, ftDate, ftTime,
  ftDateTime, ftString, ftWideString kind of fields via GetRowFieldData()
function Locate(const KeyFields: string; const KeyValues: Variant; Options: TLocateOptions) : boolean; override;

Searching a dataset for a specified record and making it the active record
- will call SearchForField protected virtual method for actual lookup

TDocVariantArrayDataSet = class(TSynVirtualDataSet)

Read-only virtual TDataSet able to access a dynamic array of TDocVariant
- could be used e.g. from the result of TMongoCollection.FindDocs() to avoid most temporary conversion into JSON or TClientDataSet buffers

constructor Create(Owner: TComponent; const Data: TVariantDynArray; const ColumnNames: array of RawUTF8; const ColumnTypes: array of TSQLDBFieldType);
reintroduce;

Initialize the virtual TDataSet from a dynamic array of TDocVariant
- you can set the expected column names and types matching the results document layout - if no column information is specified, the first TDocVariant will be used as reference

Types implemented in the SynVirtualDataSet unit

TBCDBuffer = array[0..66] of AnsiChar;

A string buffer, used by InternalBCDToBuffer to store its output text

TRecordBuffer = PChar;

Defined as TRecordBuffer = PByte in newer DB.pas

Constants implemented in the SynVirtualDataSet unit

ftDefaultMemo = ftMemo;

Map the best ft*Memo type available, depending on the Delphi compiler version

ftDefaultVCLString = ftString;

Map the VCL string type, depending on the Delphi compiler version

Functions or procedures implemented in the SynVirtualDataSet unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddBcd</td>
<td>Append a TBcd value as text to the output buffer</td>
<td>1842</td>
</tr>
<tr>
<td>BCDToCurr</td>
<td>Convert a TBcd value into a currency</td>
<td>1842</td>
</tr>
<tr>
<td>BCDToString</td>
<td>Convert a TBcd value into a VCL string text</td>
<td>1842</td>
</tr>
<tr>
<td>BCDToUTF8</td>
<td>Convert a TBcd value into a RawUTF8 text</td>
<td>1842</td>
</tr>
<tr>
<td>BCDToUTF8</td>
<td>Convert a TBcd value into a RawUTF8 text</td>
<td>1842</td>
</tr>
<tr>
<td>DataSetToJSON</td>
<td>Export all rows of a TDataSet into JSON</td>
<td>1842</td>
</tr>
<tr>
<td>InternalBCDToBuffer</td>
<td>Convert a TBcd value as text to the output buffer</td>
<td>1842</td>
</tr>
<tr>
<td>ToDataSet</td>
<td>Convert a dynamic array of TDocVariant result into a VCL DataSet</td>
<td>1842</td>
</tr>
</tbody>
</table>
procedure AddBcd(WR: TTextWriter; const AValue: TBcd);
  Append a TBcd value as text to the output buffer
  - very optimized for speed

function BCDToCurr(const AValue: TBcd; var Curr: Currency): boolean;
  Convert a TBcd value into a currency
  - purepascal version included in latest Delphi versions is slower than this

function BCDToString(const AValue: TBcd): string;
  Convert a TBcd value into a VCL string text
  - will call fast InternalBCDToBuffer function

function BCDToUTF8(const AValue: TBcd): RawUTF8; overload;
  Convert a TBcd value into a RawUTF8 text
  - will call fast InternalBCDToBuffer function

procedure BCDToUTF8(const AValue: TBcd; var result: RawUTF8); overload;
  Convert a TBcd value into a RawUTF8 text
  - will call fast InternalBCDToBuffer function

function DataSetToJSON(Data: TDataSet): RawUTF8;
  Export all rows of a TDataSet into JSON
  - will work for any kind of TDataSet

function InternalBCDToBuffer(const AValue: TBcd; out ADest: TBCDBuffer; var PBeg: PAnsiChar): integer;
  Convert a TBcd value as text to the output buffer
  - buffer is to be array[0..66] of AnsiChar
  - returns the resulting text start in PBeg, and the length as function result
  - does not handle negative sign and 0 value - see AddBcd() function use case
  - very optimized for speed

function ToDataSet(aOwner: TComponent; const Data: TVariantDynArray; const ColumnNames: array of RawUTF8; const ColumnTypes: array of TSQLDBFieldType): TDocVariantArrayDataSet; overload;
  Convert a dynamic array of TDocVariant result into a VCL DataSet
  - this function is just a wrapper around TDocVariantArrayDataSet.Create()
  - the TDataSet will be opened once created
27.44. SynWinSock.pas unit

**Purpose**: Low level access to network Sockets for the Win32 platform
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

![SynWinSock class hierarchy](image)

### Objects implemented in the *SynWinSock* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIPv6_mreq</td>
<td>IPv6 multicast address</td>
<td>1843</td>
</tr>
<tr>
<td>TPollFD</td>
<td>Polling request data structure for poll/WSAPoll</td>
<td>1843</td>
</tr>
</tbody>
</table>

**TIPv6_mreq = record**

- ipv6mr_interface: integer;
  - *IPv6 multicast address*.
- padding: integer;
  - *Interface index*.

**TPollFD = record**

- events: SHORT;
  - *Types of events poller cares about* - mainly POLLIN and/or POLLOUT
- fd: TSocket;
  - *File descriptor to poll*
- revents: SHORT;
  - *Types of events that actually occurred* - caller could just reset revents := 0 to reuse the structure

### Constants implemented in the *SynWinSock* unit

- AI_CANONNAME = $2;
  - *Socket address will be used in bind() call.*
- AI_NUMERICHOST = $4;
Return canonical name in first ai_canonname.

AI_PASSIVE = $1;

POLLERR = $0001;
Poll/WSAPoll flag error condition (always implicitly polled for)

POLLHUP = $0002;
Poll/WSAPoll flag hung up (always implicitly polled for)

POLLIN = POLLRDNORM or POLLRDBAND;
Poll/WSAPoll flag when there is data to read

POLLNVAL = $0004;
Poll/WSAPoll flag invalid polling request (always implicitly polled for)

POLLOUT = $0010;
Poll/WSAPoll flag when writing now will not block

POLLPRI = $0400;
Poll/WSAPoll flag when there is urgent data to read

POLLRDBAND = $0200;
Poll/WSAPoll flag when priority data may be read

POLLRDNORM = $0100;

POLLWRBAND = $0020;
Poll/WSAPoll flag when priority data may be written

POLLWRNORM = $0010;
Poll/WSAPoll flag when writing now will not block

Functions or procedures implemented in the SynWinSock unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>poll</td>
<td>Poll the file descriptors described by the NFDS structures starting at fds</td>
<td>1844</td>
</tr>
</tbody>
</table>

function poll(fds: PPollFD; nfds, timeout: integer): integer;
Poll the file descriptors described by the NFDS structures starting at fds
- under Windows, will call WSAPoll() emulation API - see https://blogs.msdn.microsoft.com/wndp/2006/10/26
- if TIMEOUT is nonzero and not -1, allow TIMEOUT milliseconds for an event to occur; if TIMEOUT is -1, block until an event occurs
- returns the number of file descriptors with events, zero if timed out, or -1 for errors
- before Vista, will return -1 since the API extension was not yet defined
- in practice, this API is actually slightly SLOWER than optimized Select() :(
27.45. SynZip.pas unit

**Purpose**: Low-level access to ZLib compression (1.2.5 engine version)
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Objects implemented in the SynZip unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESynZipException</td>
<td>Exception raised internally in case of Zip errors</td>
<td>1846</td>
</tr>
<tr>
<td>TFileInfo</td>
<td>Directory file information structure, as used in .zip file format</td>
<td>1846</td>
</tr>
<tr>
<td>TGZRead</td>
<td>Generic file information structure, as used in .zip file format</td>
<td>1846</td>
</tr>
<tr>
<td>TLocalFileHeader</td>
<td>Simple wrapper class to decompress a .gz file into memory or stream/file</td>
<td>1848</td>
</tr>
<tr>
<td>TLastHeader</td>
<td>Last header structure, as used in .zip file format</td>
<td>1847</td>
</tr>
<tr>
<td>TStream</td>
<td>A simple TStream descendant for compressing data into a stream</td>
<td>1847</td>
</tr>
<tr>
<td>TStream</td>
<td>Stores an entry of a file inside a .zip archive</td>
<td>1849</td>
</tr>
<tr>
<td>TLocalFileHeader</td>
<td>Read-only access to a .zip archive file</td>
<td>1849</td>
</tr>
<tr>
<td>TSynZipCompressor</td>
<td>Write-only access for creating a .zip archive file</td>
<td>1851</td>
</tr>
<tr>
<td>TZipEntry</td>
<td>Abstract write-only access for creating a .zip archive</td>
<td>1851</td>
</tr>
</tbody>
</table>
ESynZipException = class(Exception)
   Exception raised internally in case of Zip errors

TZStream = record
   The internal memory structure as expected by the ZLib library

TFileInfo = object(TObject)
   Generic file information structure, as used in .zip file format
   - used in any header, contains info about following block

extraLen: word;
   Length(name)

flags: word;
   14

nameLen: word;
   Size of uncompressed data

zCRC32: dword;
   Time in dos format

zfullSize: dword;
   Size of compressed data

zlastMod: integer;
   0=Z_STORED 8=Z_DEFLATED 12=BZ2 14=LZMA

zipMethod: word;
   0

zipSize: dword;
   Crc32 checksum of uncompressed data

function SameAs(aInfo: PFileInfo): boolean;
   0

procedure SetAlgoID(Algorithm: integer);
   1..15  (1=SynLZ e.g.) from flags

TFileHeader = object(TObject)
   Directory file information structure, as used in .zip file format
   - used at the end of the zip file to recap all entries
extFileAttr: dword;
  0 = binary; 1 = text

 fileInfo: TFileInfo;
  14

firstDiskNo: word;
  0

intFileAttr: word;
  0

localHeadOff: dword;
  "Dos file attributes"

  madeBy: word;
  \2014b50 PK#1#2

function IsFolder: boolean;
  @TLocalFileHeader

TLocalFileHeader = object(TObject)
  "Internal file information structure, as used in .zip file format"
  - used locally inside the file stream, followed by the name and then the data

  fileInfo: TFileInfo;
  \4034b50 PK#3#4

TLastHeader = record
  "Last header structure, as used in .zip file format"
  - this header ends the file and is used to find the TFileHeader entries

  commentLen: word;
    @TFileHeader

  headerDisk: word;
    0

  headerOffset: dword;
    "SizeOf(TFileHeaders + names)"

  headerSize: dword;
    1

  thisDisk: word;
    \6054b50 PK#5#6

  thisFiles: word;
    0
totalFiles: word;
1

TGZRead = object(TObject)
  Simple wrapper class to decompress a .gz file into memory or stream/file

crc32: cardinal;
  Modulo 2^32 by gzip design

function Init(gz: PAnsiChar; gzLen: ZipPtrInt): boolean;
  Read and validate the .gz header
  - on success, return true and fill complen/uncomplen/crc32c properties

functionToFile(const filename: TFileName; tempBufSize: integer=0): boolean;
  Uncompress the .gz content into a file

function ToMem: ZipString;
  Uncompress the .gz content into a memory buffer
  - warning: won't work as expected if uncomplen32 was truncated to 2^32

function ToStream(stream: TStream; tempBufSize: integer=0): boolean;
  Uncompress the .gz content into a stream

function ZStreamDone: boolean;
  Any successful call to ZStreamStart should always run ZStreamDone
  - return true if the crc and the uncompressed size are ok

function ZStreamNext: integer;
  Will uncompress into dest/destsize buffer as supplied to ZStreamStart
  - return the number of bytes uncompressed (<=destsize)
  - return 0 if the input stream is finished

function ZStreamStart(dest: pointer; destsize: integer): boolean;
  Allow low level iterative decompression using an internal TZStream structure

function ZStreamStarted: boolean;
  Return true if ZStreamStart() has been successfully called

TSynZipCompressor = class(TStream)
  A simple TStream descendant for compressing data into a stream
  - this simple version don't use any internal buffer, but rely on Zip library buffering system
  - the version in SynZipFiles is much more powerfull, but this one is sufficient for most common cases (e.g. for on the fly .gz backup)

constructor Create(outStream: TStream; CompressionLevel: Integer; Format: TSynZipCompressorFormat = szcfRaw);
  Create a compression stream, writting the compressed data into the specified stream (e.g. a file stream)

destructor Destroy; override;
  Release memory
function Read(var Buffer; Count: Longint): Longint; override;
   This method will raise an error: it's a compression-only stream

function Seek(Offset: Longint; Origin: Word): Longint; override;
   Used to return the current position, i.e. the real byte written count
   - for real seek, this method will raise an error: it's a compression-only stream

function SizeIn: cardinal;
   The number of byte written, i.e. the current uncompressed size

function SizeOut: cardinal;
   The number of byte sent to the destination stream, i.e. the current compressed size

function Write(const Buffer; Count: Longint): Longint; override;
   Add some data to be compressed

procedure Flush;
   Write all pending compressed data into outStream

property CRC: cardinal read fCRC;
   The current CRC of the written data, i.e. the uncompressed data CRC

TZipEntry = record
   Stores an entry of a file inside a .zip archive
   data: PAnsiChar;
      Points to the compressed data in the .zip archive, mapped in memory
   infoDirectory: PFileHeader;
      The information of this file, as stored at the end of the .zip archive
      - may differ from infoLocal^ content, depending of the zipper tool used
   infoLocal: PFileInfo;
      The information of this file, as stored locally in the .zip archive
      - note that infoLocal^.zipSize/zfullSize/zcrc32 may be 0 if the info was stored in a "data descriptor" block after the data: in this case, you should use TZipRead.RetrieveFileInfo() instead of this structure
   storedName: PAnsiChar;
      Name of the file inside the .zip archive
      - not ASCIIZ: length = infoLocal.nameLen
   zipName: TFileName;
      Name of the file inside the .zip archive
      - converted from DOS/OEM or UTF-8 into generic (Unicode) string

TZipRead = class(TObject)
   Read-only access to a .zip archive file
   - can open directly a specified .zip file (will be memory mapped for fast access)
   - can open a .zip archive file content from a resource (embedded in the executable)
   - can open a .zip archive file content from memory
Count: integer;
\textit{The number of files inside a .zip archive}

Entry: array of TZipEntry;
\textit{The files inside the .zip archive}

\textbf{constructor} Create(aFile: THandle; ZipStartOffset: cardinal=0; Size: cardinal=0); overload;
\textit{Open a .zip archive file from its File Handle}

\textbf{constructor} Create(BufZip: PByteArray; Size: cardinal); overload;
\textit{Open a .zip archive file directly from memory}

\textbf{constructor} Create(const aFileName: TFileName; ZipStartOffset: cardinal=0; Size: cardinal=0); overload;
\textit{Open a .zip archive file as Read Only}

\textbf{constructor} Create(Instance: THandle; const ResName: string; ResType: PChar); overload;
\textit{Open a .zip archive file directly from a resource}

\textbf{destructor} Destroy; \textit{override};
\textit{Release associated memory}

\textbf{function} NameToIndex(const aName: TFileName): integer;
\textit{Get the index of a file inside the .zip archive}

\textbf{function} RetrieveFileInfo(Index: integer; \textbf{var} Info: TFileInfo): boolean;
\textit{Retrieve information about a file}
\quad - in some cases (e.g. for a .zip created by latest Java JRE), infoLocal^.zzipSize/zfullSize/zcrc32 may equal 0: this method is able to retrieve the information either from the ending "central directory", or by searching the "data descriptor" block
\quad - returns TRUE if the Index is correct and the info was retrieved
\quad - returns FALSE if the information was not successfully retrieved

\textbf{function} UnZip(const aName: TFileName): ZipString; overload;
\textit{Uncompress a file stored inside the .zip archive into memory}

\textbf{function} UnZip(const aName, DestDir: TFileName; DestDirIsFileName: boolean=false): boolean; overload;
\textit{Uncompress a file stored inside the .zip archive into a destination directory}

\textbf{function} UnZip(aIndex: integer): ZipString; overload;
\textit{Uncompress a file stored inside the .zip archive into memory}

\textbf{function} UnZip(aIndex: integer; aDest: TStream): boolean; overload;
\textit{Uncompress a file stored inside the .zip archive into a stream}

\textbf{function} UnZip(aIndex: integer; const DestDir: TFileName; DestDirIsFileName: boolean=false): boolean; overload;
\textit{Uncompress a file stored inside the .zip archive into a destination directory}

\textbf{function} UnZipAll(DestDir: TFileName): integer;
\textit{Uncompress all fields stored inside the .zip archive into the supplied destination directory}
\quad - returns -1 on success, or the index in Entry[] of the failing file
TZipWriteAbstract = class(TObject)
Abstract write-only access for creating a .zip archive

Count: integer;
The total number of entries

Entry: array of record intName: ZipString; fhr: TFileHeader; end;
The resulting file entries, ready to be written as a .zip catalog
- those will be appended after the data blocks at the end of the .zip file

constructor Create;
The file name, as stored in the .zip internal directory the corresponding file header initialize the .zip archive
- a new .zip file content is prepared

destructor Destroy; override;
Release associated memory, and close destination archive

procedure AddDeflated(const aZipName: TFileName; Buf: pointer; Size: integer; CompressLevel: integer=6; FileAge: integer=1+1 shl 5+30 shl 9); overload;
Compress (using the deflate method) a memory buffer, and add it to the zip file
- by default, the 1st of January, 2010 is used if not date is supplied

procedure AddStored(const aZipName: TFileName; Buf: pointer; Size: integer; FileAge: integer=1+1 shl 5+30 shl 9);
Add a memory buffer to the zip file, without compression
- content is stored, not deflated (in that case, no deflate code is added to the executable)
- by default, the 1st of January, 2010 is used if not date is supplied

procedure Append(const Content: ZipString);
Append a file content into the destination file
- useful to add the initial Setup.exe file, e.g.

TZipWrite = class(TZipWriteAbstract)
Write-only access for creating a .zip archive file
- not to be used to update a .zip file, but to create a new one
- update can be done manually by using a TZipRead instance and the AddFromZip() method

Handle: integer;
The associated file handle

constructor Create(const aFileName: TFileName); overload;
Initialize the .zip file
- a new .zip file content is created

constructor CreateFrom(const aFileName: TFileName; dummy: integer=0);
Initialize an existing .zip file in order to add some content to it
- warning: AddStored/AddDeflated() won't check for duplicate zip entries
- this method is very fast, and will increase the .zip file in-place (the old content is not copied, new data is appended at the file end)
- "dummy" parameter exists only to disambiguate constructors for C++
```pascal

destructor Destroy; override;
  Release associated memory, and close destination file

procedure AddDeflated(const aFileName: TFileName; RemovePath: boolean=true;
  CompressLevel: integer=6; ZipName: TFileName=''); overload;
  Compress (using the deflate method) a file, and add it to the zip file

procedure AddFolder(const FolderName: TFileName; const Mask:
  TFileName=ZIP_FILES_ALL; Recursive: boolean=true; CompressLevel: integer=6);
  Compress (using the deflate method) all files within a folder, and add it to the zip file
  - if Recursive is TRUE, would include files from nested sub-folders

procedure AddFromZip(const ZipEntry: TZipEntry);
  Add a file from an already compressed zip entry

TZipWriteToStream = class(TZipWriteAbstract)
  Write-only access for creating a .zip archive into a stream

constructor Create(aDest: TStream);
  Initialize the .zip archive
  - a new .zip file content is prepared

Types implemented in the SynZip unit

PLastHeader = ^TLastHeader;
0

TSynZipCompressorFormat = ( szcfRaw, szcfZip, szcfGZ );
  The format used for storing data

TZLong = cardinal;
  Statically linked with old 32-bit TZStream

ZipPtrUInt = cardinal;
  As available in FPC

ZipString = type AnsiString;
  Define a raw storage string type, used for data buffer management

Constants implemented in the SynZip unit

DEF_MEM_LEVEL = 8;
  32K LZ77 window

ZIP_FILES_ALL = '*.*';
  Operating-system dependent wildcard to match all files in a folder

Functions or procedures implemented in the SynZip unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check</td>
<td>Low-level check of the code returned by the ZLib library</td>
<td>1853</td>
</tr>
</tbody>
</table>
```

SynZip.pas unit - Rev. 1.18
### Functions or procedures

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompressDeflate</td>
<td>(un)compress a data content using the Deflate algorithm (i.e. &quot;raw deflate&quot;)</td>
<td>1853</td>
</tr>
<tr>
<td>CompressGZip</td>
<td>(un)compress a data content using the gzip algorithm</td>
<td>1854</td>
</tr>
<tr>
<td>CompressMem</td>
<td>In-memory ZLib DEFLATE compression</td>
<td>1854</td>
</tr>
<tr>
<td>CompressStream</td>
<td>ZLib DEFLATE compression from memory into a stream</td>
<td>1854</td>
</tr>
<tr>
<td>CompressString</td>
<td>Compress some data, with a proprietary format (including CRC)</td>
<td>1854</td>
</tr>
<tr>
<td>CompressZLib</td>
<td>(un)compress a data content using the zlib algorithm</td>
<td>1854</td>
</tr>
<tr>
<td>CRC32string</td>
<td>Just hash aString with CRC32 algorithm</td>
<td>1854</td>
</tr>
<tr>
<td>DeflateInit</td>
<td>Prepare the internal memory structure as expected by the ZLib library for compression</td>
<td>1854</td>
</tr>
<tr>
<td>EventArchiveZip</td>
<td>A TSynLogArchiveEvent handler which will compress older .log files into .zip archive files</td>
<td>1854</td>
</tr>
<tr>
<td>GZFile</td>
<td>Compress a file content into a new .gz file</td>
<td>1854</td>
</tr>
<tr>
<td>GZRead</td>
<td>Uncompress a .gz file content</td>
<td>1854</td>
</tr>
<tr>
<td>StreamInit</td>
<td>Initialize the internal memory structure as expected by the ZLib library</td>
<td>1855</td>
</tr>
<tr>
<td>UnCompressMem</td>
<td>In-memory ZLib INFLATE decompression</td>
<td>1855</td>
</tr>
<tr>
<td>UnCompressStream</td>
<td>ZLib INFLATE decompression from memory into a stream</td>
<td>1855</td>
</tr>
<tr>
<td>UncompressString</td>
<td>Uncompress some data, with a proprietary format (including CRC)</td>
<td>1855</td>
</tr>
<tr>
<td>UnCompressZipString</td>
<td>ZLib INFLATE decompression from memory into a AnsiString (ZipString) variable</td>
<td>1855</td>
</tr>
</tbody>
</table>

```FUNCTION Check(const Code: Integer; const ValidCodes: array of Integer; const Context: string=''): integer;
Low-level check of the code returned by the ZLib library

FUNCTION CompressDeflate(var DataRawByteString; Compress: boolean): AnsiString;
(un)compress a data content using the Deflate algorithm (i.e. "raw deflate")
- as expected by THttpSocket.RegisterCompress
- will use internaly a level compression of 1, i.e. fastest available (content of 4803 bytes is compressed into 700, and time is 440 us instead of 220 us)
- deflate content encoding is pretty inconsistent in practice, so slightly slower CompressGZip() is preferred - http://stackoverflow.com/a/9186091```
function CompressGZip(var DataRawByteString; Compress: boolean): AnsiString;

(un)compress a data content using the gzip algorithm
- as expected by THttpSocket.RegisterCompress
- will use internal a level compression of 1, i.e. fastest available (content of 4803 bytes is compressed into 700, and time is 440 us instead of 220 us)

function CompressMem(src, dst: pointer; srcLen, dstLen: integer; CompressionLevel: integer=6; ZlibFormat: Boolean=false): integer;

In-memory ZLib DEFLATE compression
- by default, will use the deflate/.zip header-less format, but you may set ZlibFormat=true to add an header, as expected by zlib (and pdf)

function CompressStream(src: pointer; srcLen: integer; tmp: TStream; CompressionLevel: integer=6; ZlibFormat: Boolean=false; TempBufSize: integer=0): cardinal;

ZLib DEFLATE compression from memory into a stream
- by default, will use the deflate/.zip header-less format, but you may set ZlibFormat=true to add an header, as expected by zlib (and pdf)

function CompressString(const data: ZipString; failIfGrow: boolean = false; CompressionLevel: integer=6): ZipString;

Compress some data, with a proprietary format (including CRC)

function CompressZLib(var DataRawByteString; Compress: boolean): AnsiString;

(un)compress a data content using the zlib algorithm
- as expected by THttpSocket.RegisterCompress
- will use internal a level compression of 1, i.e. fastest available (content of 4803 bytes is compressed into 700, and time is 440 us instead of 220 us)
- zlib content encoding is pretty inconsistent in practice, so slightly slower CompressGZip() is preferred - http://stackoverflow.com/a/9186091

function CRC32string(const aString: ZipString): cardinal;

Just hash aString with CRC32 algorithm
- crc32 is better than adler32 for short strings

function DeflateInit(var Stream: TZStream; CompressionLevel: integer; ZlibFormat: Boolean): Boolean; overload;

Prepare the internal memory structure as expected by the ZLib library for compression

function EventArchiveZip(const aOldLogFileName, aDestinationPath: TFileName): boolean;

A TSynLogArchiveEvent handler which will compress older .log files into .zip archive files
- resulting file will be named YYYYMM.zip and will be located in the aDestinationPath directory, i.e. TSynLogFamily.ArchivePath+'\log\YYYYMM.zip'

function GZFile(const orig, destgz: TFileName; CompressionLevel: Integer=6): boolean;

Compress a file content into a new .gz file
- will use TSynZipCompressor for minimal memory use during file compression

function GZRead(gz: PAnsiChar; gzLen: integer): ZipString;

Uncompress a .gz file content
- return "" if the .gz content is invalid (e.g. bad crc)
procedure StreamInit(var Stream: TZStream); overload;

Initialize the internal memory structure as expected by the ZLib library

function UnCompressMem(src, dst: pointer; srcLen, dstLen: integer; ZlibFormat: Boolean=false) : integer;

In-memory ZLib INFLATE decompression
- by default, will use the deflate/.zip header-less format, but you may set ZlibFormat=true to add an header, as expected by zlib (and pdf)

function UnCompressStream(src: pointer; srcLen: integer; tmp: TStream; checkCRC: PCardinal; ZlibFormat: Boolean=false; TempBufSize: integer=0): cardinal;

ZLib INFLATE decompression from memory into a stream
- return the number of bytes written into the stream
- if checkCRC if not nil, it will contain the crc32; if aStream is nil, it will only calculate the crc of the the uncompressed memory block
- by default, will use the deflate/.zip header-less format, but you may set ZlibFormat=true to add an header, as expected by zlib (and pdf)

function UncompressString(const data: ZipString) : ZipString;

Uncompress some data, with a proprietary format (including CRC)
- return '' in case of a decompression failure

function UnCompressZipString(src: pointer; srcLen: integer; out data: ZipString; checkCRC: PCardinal; ZlibFormat: Boolean; TempBufSize: integer=0): cardinal;

ZLib INFLATE decompression from memory into a AnsiString (ZipString) variable
- return the number of bytes written into the string
- if checkCRC if not nil, it will contain the crc32; if aStream is nil, it will only calculate the crc of the the uncompressed memory block
- by default, will use the deflate/.zip header-less format, but you may set ZlibFormat=true to add an header, as expected by zlib (and pdf)
27.46. SynZipFiles.pas unit

Purpose: High-level access to .zip archive file compression
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the SynZipFiles unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynZip</td>
<td>Low-level access to ZLib compression (1.2.5 engine version)</td>
<td>1845</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

SynZipFiles class hierarchy

Objects implemented in the SynZipFiles unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSynCompressionAlgoBuf</td>
<td>Template class for 64KB chunked (not whole) algorithm (SynLZ, LZO...) which forces storing as uncompressed if compression ratio has no gain</td>
<td>1857</td>
</tr>
<tr>
<td>TSynCompressionAlgoWhole</td>
<td>Template class for whole algorithm (SynLZ, LZO...) which forces storing as uncompressed if compression ratio has no gain</td>
<td>1857</td>
</tr>
<tr>
<td>TZip</td>
<td>TZip handles ZIP standard files on disk</td>
<td>1858</td>
</tr>
</tbody>
</table>
Objects | Description | Page
--- | --- | ---
TZipReader | Databuf: AnsiChar | 1857

**TSynCompressionAlgoBuf = class(TSynCompressionAlgo)**

Template class for 64KB chunked (not whole) algorithm (SynLZ, LZO...) which forces storing as uncompressed if compression ratio has no gain

**Destructor** Destroy; **override;**

Free fCompressBuf memory if allocated

**Function** Compress(InP: pointer; InLen: cardinal; CRC: PCardinal): cardinal; **override;**

Compress InP[InLen] into OutStream + update CRC, return compressed length

**Function** UnCompress(InP: pointer; InLen: cardinal; OutP: pointer): cardinal; **override;**

Uncompress InP[InLen] into OutP, return uncompressed length

**Function** UnCompressedLength(InP: pointer; InLen: cardinal): cardinal; **override;**

Return uncompressed length of InP[InLen] for proper mem allocation

**Procedure** CompressInit(OutStream: TStream); **override;**

Initialize compression into OutStream

**TSynCompressionAlgoWhole = class(TSynCompressionAlgo)**

Template class for whole algorithm (SynLZ, LZO...) which forces storing as uncompressed if compression ratio has no gain

**Function** Compress(InP: pointer; InLen: cardinal; CRC: PCardinal): cardinal; **override;**

Compress InP[InLen] into OutStream + update CRC, return compressed length

**Function** UnCompress(InP: pointer; InLen: cardinal; OutP: pointer): cardinal; **override;**

Uncompress InP[InLen] into OutP, return uncompressed length

**Function** UnCompressedLength(InP: pointer; InLen: cardinal): cardinal; **override;**

Return uncompressed length of InP[InLen] for proper mem allocation

**TZipReader = class(TZipCommon)**

Databuf: AnsiChar

**Function** GetData(aIndex: integer; aStream: TStream=nil; CheckCRC: boolean=false; asBlobDataStored: boolean=false; withAlgoDataLen: boolean=false): PAnsiChar;

Force Count=0

**Function** SameAs(aReader: TZipReader): boolean;

Save uncompressed to stream
procedure DeleteLastEntry;
    TBlobData->aStream

procedure GetBlobData(aIndex: integer; aStream: TStream); overload;
    PBlobData(result)

procedure SaveToStream(aStream: TStream);
    Don't use inside TZipValues: already done in Create

TZip = class(TObject)
    TZip handles ZIP standard files on disk

function MarkDeletedBefore(aDate: TDateTime; aBackup: TZip=nil): boolean;
    Before any ZipCreate

function SameAs(aZip: TZip): boolean;
    Flushed at Destroy

procedure ZipClose;
    Use Zip.Write() to send data before ZipClose

Types implemented in the SynZipFiles unit

PBlobData = ^TBlobData;
    Used to transfer Blob Data from/to Client without compress/uncompress:

Constants implemented in the SynZipFiles unit

BLOBDATA_HEADSIZE = sizeof(TBlobData)-sizeof(AnsiChar);
    Points to next bloc

Functions or procedures implemented in the SynZipFiles unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GZRead</td>
<td>Create a TBlobData in aStream - can use encryption with algo 7=AES+Zip-chunked and 8=AES+SynLz-chunked</td>
<td>1858</td>
</tr>
</tbody>
</table>

function GZRead(const aFileName: TFileName): RawByteString; overload;
    Create a TBlobData in aStream - can use encryption with algo 7=AES+Zip-chunked and 8=AES+SynLz-chunked

Variables implemented in the SynZipFiles unit

BlobDataNull: TBlobData;
    Before any AddValue
27.47. SynCrossPlatformCrypto.pas unit

**Purpose:** Cryptographic cross-platform units
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Objects implemented in the *SynCrossPlatformCrypto* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSHA256</td>
<td>Class for SHA256 hashing</td>
<td>1859</td>
</tr>
<tr>
<td>TSHAHash</td>
<td>Internal work buffer for SHA256 hashing</td>
<td>1859</td>
</tr>
</tbody>
</table>

```pascal
TSHAHash = record
  Internal work buffer for SHA256 hashing
```

```pascal
TSHA256 = class(TObject)
  Class for SHA256 hashing
  constructor Create;
    Initialize SHA256 context for hashing
  function Finalize: string;
    Finalize and compute the resulting SHA256 hash Digest of all data affected to Update() method
    - returns the data as Hexadecimal
  procedure Update(const ascii: string); overload;
    Update the SHA256 context with 8 bit ascii data (e.g. UTF-8)
  procedure Update(const buf: array of byte); overload;
    Update the SHA256 context with some data
```

Types implemented in the *SynCrossPlatformCrypto* unit

```pascal
TSHA256Buffer = array[0..63] of hash32;
  Internal buffer for SHA256 hashing
```

Functions or procedures implemented in the *SynCrossPlatformCrypto* unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc32</td>
<td>Compute the zlib/deflate crc32 hash value on a supplied buffer</td>
<td>1860</td>
</tr>
<tr>
<td>crc32ascii</td>
<td>Compute the zlib/deflate crc32 hash value on a supplied ASCII-7 buffer</td>
<td>1860</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>SHA256</td>
<td>Compute SHA256 hexa digest of a supplied buffer</td>
<td>1860</td>
</tr>
<tr>
<td>SHA256</td>
<td>Compute SHA256 hexa digest of a supplied 8 bit ascii data (e.g. UTF-8)</td>
<td>1860</td>
</tr>
</tbody>
</table>

```pascal
function crc32(aCrc32: hash32; const buf: array of byte): hash32;
  Compute the zlib/deflate crc32 hash value on a supplied buffer
end;
```

```pascal
function crc32ascii(aCrc32: hash32; const buf: string): hash32;
  Compute the zlib/deflate crc32 hash value on a supplied ASCII-7 buffer
end;
```

```pascal
function SHA256(const buf: string): string; overload;
  Compute SHA256 hexa digest of a supplied 8 bit ascii data (e.g. UTF-8)
end;
```

```pascal
function SHA256(const buf: array of byte): string; overload;
  Compute SHA256 hexa digest of a supplied buffer
end;
```

Variables implemented in the `SynCrossPlatformCrypto` unit

```pascal
crc32tab: array[byte] of hash32;
```

*Table used by crc32() function*
- Table content is created from code in initialization section below
27.48. SynCrossPlatformJSON.pas unit

*Purpose*: Minimum standand-alone cross-platform JSON process using variants
- This unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

```
Exception
    EJSONException

TInvokeableVariantType
    TJSONVariant

TObject
    TJSONVariantData
    TJSONTable
    TJSONTableObject

SynCrossPlatformJSON class hierarchy
```

**Objects implemented in the SynCrossPlatformJSON unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJSONException</td>
<td>Exception used during standand-alone cross-platform JSON process</td>
<td>1861</td>
</tr>
<tr>
<td>TJSONTable</td>
<td>Handle a JSON result table, as returned by mORMot's server</td>
<td>1863</td>
</tr>
<tr>
<td>TJSONTableObject</td>
<td>Handle a JSON result table, as returned by mORMot's server</td>
<td>1864</td>
</tr>
<tr>
<td>TJSONVariant</td>
<td>Low-level class used to register TJSONVariantData as custom type</td>
<td>1863</td>
</tr>
<tr>
<td>TJSONVariantData</td>
<td>Stores any JSON object or array as variant</td>
<td>1861</td>
</tr>
<tr>
<td>TPublishedMethod</td>
<td>Used e.g. by TSynTest for each test case</td>
<td>1864</td>
</tr>
</tbody>
</table>

**EJSONException = class(Exception)**

*Exception used during standand-alone cross-platform JSON process*

**TJSONVariantData = object(TObject)**

*Stores any JSON object or array as variant*
- This structure is not very optimized for speed or memory use, but is simple and strong enough for our client-side purpose
- It is in fact already faster (and using less memory) than DBXJSON and SuperObject / XSuperObject libraries - of course, mORMot's TDocVariant is faster, as dwsJSON is in some cases, but those are not cross-platform

*Names: TStringDynArray;*

*Names of this jvObject*

*Values: TVariantDynArray;*

*Values of this jvObject or jvArray*
function AddItem: PJSONVariantData;
  Add a void TJSONVariantData to the jvArray and return a pointer to it

function Data(const aName: string): PJSONVariantData;
  Access to a nested TJSONVariantData item
  - returns nil if aName was not found, or not a true TJSONVariantData item

function EnsureData(const aPath: string): PJSONVariantData;
  Access to a nested TJSONVariantData item, creating it if necessary
  - aPath can be specified with any depth, e.g. 'level1.level2.level3'
  - if the item does not exist or is not a true TJSONVariantData, a new one will be created, and returned as pointer

function FromJSON(const JSON: string): boolean;
  Fill this document from a JSON array or object

function NameIndex(const aName: string): integer;
  Search for a name in this jvObject

function ToJSON: string;
  Convert this document into JSON array or object

function ToNewObject: TObject;
  Create an instance, and fill its published properties from this JSON object
  - it should contain some "ClassName" properties, i.e. JSON should have been created by ObjectToJSON(Instance,true) and the class should have been registered with RegisterClassForJSON()
procedure InitFrom(const aValues: TVariantDynArray); overload;
  Initialize the low-level memory structure with a given array of variant
  - you should call Clear before calling overloaded Init several times

procedure SetPath(const aPath: string; const aValue: variant);
  Set a value of this jvObject to a given path
  - aPath can be specified with any depth, e.g. 'level1.level2.level3'

property Count: integer read GetCount;
  Number of items in this jvObject or jvArray
  - returns 0 if this instance is not a TJSONVariant custom variant

property Item[aIndex: integer]: variant read GetItem write SetItem;
  Access by index to a value of this jvArray
  - will return UnAssigned if aindex is not correct or this is not a jvArray

property Kind: TJSONVariantKind read GetKind;
  Kind of document this TJSONVariantData contains
  - returns jvUndefined if this instance is not a TJSONVariant custom variant

property Value[const aName: string]: variant read GetValue write SetValue;
  Access by name to a value of this jvObject
  - value is returned as (varVariant or varByRef) for best speed
  - will return UnAssigned if aName is not correct or this is not a jvObject

property ValueCopy[const aName: string]: variant read GetValueCopy;
  Access by name to a value of this jvObject
  - value is returned as a true copy (not varByRef) so this property is slower but safer than Value[],
    if the owning TJsonVariantData disappears
  - will return UnAssigned if aName is not correct or this is not a jvObject

TJSONVariant = class(TInvokeableVariantType)
  Low-level class used to register TJSONVariantData as custom type
  - allows late binding to values, e.g.
    jsonvar.avalue := jsonvar.avalue + 1;
  - due to an issue with FPC implementation, you can only read properties, not set them, so you
    should write:
    TJSONVariantData(jsonvar)['avalue'] := jsonvar.avalue + 1;

TJSONTable = class(TObject)
  Handle a JSON result table, as returned by mORMot's server
  - handle both expanded and non expanded layout
  - will be used e.g. on client side for variant-based ORM data parsing

constructor Create(const aJSON: string);
  Parse the supplied JSON content

function FieldIndex(const FieldName: string): integer;
  Case-insensitive search for a field name
function Step(SeekFirst: boolean=false): boolean;

To be called in a loop to iterate through all data rows
- if returned true, Value[] contains the fields of this row

function StepValue(var RowValues: variant; SeekFirst: boolean=false): boolean;

To be called in a loop to iterate through all data rows
- if returned true, RowValues contains this row as TJSONVariant

property FieldNames: TStringDynArray read fFieldNames;
The recognized field names

property JSON: string read fJSON;
The associated JSON content

property RowValues: TVariantDynArray read fRowValues;
After Step() returned true, can be used to retrieve a field value by index

property Value[const FieldName: string]: variant read Get;
After Step() returned true, can be used to retrieve a field value by name

TJSONTableObject = class(TJSONTable)
Handle a JSON result table, as returned by mORMot's server
- handle both expanded and non expanded layout
- this class is able to use RTTI to fill all published properties of a TObject

function StepObject(Instance: TObject; SeekFirst: boolean=false): boolean; virtual;
To be called in a loop to iterate through all data rows
- if returned true, Object published properties will contain this row

TPublishedMethod = record
Used e.g. by TSynTest for each test case

Types implemented in the SynCrossPlatformJSON unit

NativeInt = integer;
Delphi 2009 NativeUInt is buggy

TByteDynArray = array of byte;
This type is used to store BLOB content

TJSONVariantKind = ( jvUndefined, jvObject, jvArray );
Which kind of document the TJSONVariantData contains

TPublishedMethodDynArray = array of TPublishedMethod;
As filled by GetPublishedMethods()

TRTTIPropInfo = PPropInfo;
An abstract type used for RTTI property information

TRTTITypeInfo = PPropInfo;
An abstract type used for RTTI type information

TUTF8Buffer = UTF8String;
This type will store UTF-8 encoded buffer (also on NextGen platform)
Constants implemented in the SynCrossPlatformJSON unit

```pascal
JSON_BASE64_MAGIC: array[0..2] of byte = ($ef,$bf,$b0);
```

*Special code to mark Base64 binary content in JSON string*
- Unicode special char U+FFF0 is UTF-8 encoded as EF BF 80 bytes
- prior to Delphi 2009, it won't work as expected since U+FFF0 won't be able to be converted into U+FFFF

```pascal
JSON_BASE64_MAGIC_LEN = sizeof(JSON_BASE64_MAGIC) div sizeof(char);
```

*Size, in platform chars, of our special code to mark Base64 binary content in JSON string*
- equals 1 since Delphi 2009 (UTF-16 encoded), or 3 for older versions (UTF-8 encoded) of the compiler compiler

Functions or procedures implemented in the SynCrossPlatformJSON unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppendChar</td>
<td>This function is faster than str := str + chr !</td>
</tr>
<tr>
<td>Base64JSONStringToBytes</td>
<td>Decode a Base64-encoded string</td>
</tr>
<tr>
<td>BytesToBase64JSONString</td>
<td>Base-64 encode a BLOB into string</td>
</tr>
<tr>
<td>CreateClassForJSON</td>
<td>Create a class instance from its name</td>
</tr>
<tr>
<td>DateTimeToIso8601</td>
<td>Compute the unquoted ISO-8601 text representation of a date/time value</td>
</tr>
<tr>
<td>DateTimeToJSON</td>
<td>Compute the ISO-8601 JSON text representation of a date/time value</td>
</tr>
<tr>
<td>DoubleQuoteStr</td>
<td>Convert the supplied text as &quot;text&quot;, as expected by SQL standard</td>
</tr>
<tr>
<td>DoubleToJSON</td>
<td>Compute the JSON representation of a floating-point value</td>
</tr>
<tr>
<td>GetInstanceProp</td>
<td>Retrieve the value of a published property as variant</td>
</tr>
<tr>
<td>GetPropsInfo</td>
<td>Retrieve the published properties type information about a given class</td>
</tr>
<tr>
<td>GetPublishedMethods</td>
<td>Retrieve all the published methods of a given class, using RTTI</td>
</tr>
<tr>
<td>IdemPropName</td>
<td>Check that two ASCII-7 latin text do match</td>
</tr>
<tr>
<td>Iso8601ToDateTime</td>
<td>Convert unquoted ISO-8601 text representation into a date/time value</td>
</tr>
<tr>
<td>JSONToNewObject</td>
<td>Create a new object and fill its published properties from the supplied JSON object, which should include &quot;ClassName&quot;: &quot;...&quot; properties</td>
</tr>
<tr>
<td>JSONToObject</td>
<td>Fill an object published properties from the supplied JSON object</td>
</tr>
<tr>
<td>JSONToObjectList</td>
<td>Create a list of object published properties from the supplied JSON object</td>
</tr>
<tr>
<td>JSONToValue</td>
<td>Compute a variant from its JSON representation</td>
</tr>
</tbody>
</table>
### Functions or procedures

<table>
<thead>
<tr>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSONVariant Create a TJSONVariant instance from a given JSON content</td>
<td>1868</td>
</tr>
<tr>
<td>JSONVariant Create a TJSONVariant TJSONVariant array from a supplied array of values</td>
<td>1868</td>
</tr>
<tr>
<td>JSONVariantData Access to a TJSONVariant instance members</td>
<td>1868</td>
</tr>
<tr>
<td>JSONVariantDataSafe Access to a TJSONVariant instance members</td>
<td>1868</td>
</tr>
<tr>
<td>JSONVariantFromConst Create a TJSONVariant TJSONVariant array from a supplied array of values</td>
<td>1868</td>
</tr>
<tr>
<td>NowToIso8601 Compute the ISO-8601 JSON text representation of the current date/time value</td>
<td>1869</td>
</tr>
<tr>
<td>ObjectToJSON Compute the JSON representation of an object published properties</td>
<td>1869</td>
</tr>
<tr>
<td>RegisterClassForJSON Register the class types to be created from its name</td>
<td>1869</td>
</tr>
<tr>
<td>RTTIPropInfoTypeName Return a string corresponding to the type name, as stored in the RTTI</td>
<td>1869</td>
</tr>
<tr>
<td>SetInstanceProp Set the value of a published property from a variant</td>
<td>1869</td>
</tr>
<tr>
<td>ShortStringToString Convert ASCII-7 latin text, encoded as a shortstring buffer, into a string</td>
<td>1869</td>
</tr>
<tr>
<td>StartWithPropName Check that two ASCII-7 latin text do match</td>
<td>1869</td>
</tr>
<tr>
<td>StringToJSON Compute the quoted JSON string corresponding to the supplied text</td>
<td>1869</td>
</tr>
<tr>
<td>UTF8FileToString Read an UTF-8 (JSON) file into a native string</td>
<td>1869</td>
</tr>
<tr>
<td>ValueToJSON Compute the JSON representation of a variant value</td>
<td>1869</td>
</tr>
<tr>
<td>VarRecToValue Convert an &quot;array of const&quot; parameter value into its string representation</td>
<td>1869</td>
</tr>
</tbody>
</table>

**procedure** AppendChar(var str: string; chr: Char);

This function is faster than str := str+chr !

**function** Base64JSONStringToBytes(const JSONString: string; var Bytes: TByteDynArray; withBase64Magic: boolean=true): boolean;

Decode a Base64-encoded string
- default withBase64Magic=TRUE will expect the string to start with our JSON_BASE64_MAGIC marker

**function** BytesToBase64JSONString(const Bytes: TByteDynArray; withBase64Magic: boolean=true): string;

Base-64 encode a BLOB into string
- default withBase64Magic=TRUE will include our JSON_BASE64_MAGIC marker
function CreateClassForJSON(const ClassName: string): TObject;

Create a class instance from its name
- the class should have been registered previously via RegisterClassForJSON()
- if the supplied class name is not found, will return nil

function DateTimeToIso8601(Value: TDateTime): string;

Compute the unquoted ISO-8601 text representation of a date/time value
- e.g. 'YYYY-MM-DD' 'Thh:mm:ss' or 'YYYY-MM-DDThh:mm:ss'
- if Date is 0, will return ''

function DateTimeToJSON(Value: TDateTime): string;

Compute the ISO-8601 JSON text representation of a date/time value
- e.g. "YYYY-MM-DD" "Thh:mm:ss" or "YYYY-MM-DDThh:mm:ss"
- if Date is 0, will return ""

procedure DoubleQuoteStr(var text: string);

Convert the supplied text as "text", as expected by SQL standard

procedure DoubleToJSON(Value: double; var result: string);

Compute the JSON representation of a floating-point value

function GetInstanceProp(Instance: TObject; PropInfo: TRTTIPropInfo; StoreClassName: boolean = False): variant;

Retrieve the value of a published property as variant

procedure GetPropsInfo(TypeInfo: TRTTITypeInfo; var PropNames: TStringDynArray; var PropRTTI: TRTTIPropInfoDynArray);

Retrieve the published properties type information about a given class

procedure GetPublishedMethods(Instance: TObject; out Methods: TPublishedMethodDynArray);

Retrieve all the published methods of a given class, using RTTI

function IdemPropName(const PropName1,PropName2: string): boolean; overload;

Check that two ASCII-7 latin text do match

function IdemPropName(PropName1: PByteArray; const PropName2: string): boolean; overload;

Check that two ASCII-7 latin text do match
- first parameter is expected to be a shortstring low-level buffer - as such, this overloaded function would work with NEXTGEN encoded RTTI

function Iso8601ToDateTime(const Value: string): TDateTime;

Convert unquoted ISO-8601 text representation into a date/time value
- e.g. 'YYYY-MM-DD' 'Thh:mm:ss' or 'YYYY-MM-DDThh:mm:ss'

function JSONToNewObject(const JSON: string): pointer;

Create a new object and fill its published properties from the supplied JSON object, which should include "ClassName":"..." properties
- JSON should have been created with ObjectToJSON(Instance,true) and the class should have been registered with RegisterClassForJSON()
function JSONToObject(Instance: TObject; const JSON: string): boolean;

Fill an object published properties from the supplied JSON object
- handle only simple types of properties, not nested class instances

function JSONToObjectList(ItemClass: TClass; const JSON: string): TObjectList;

Create a list of object published properties from the supplied JSON object
- handle only simple types of properties, not nested class instances

function JSONToObjectValue(const JSON: string): variant;

Compute a variant from its JSON representation
- will work for simple types, or TJSONVariant object or array

function JSONVariant(const values: TVariantDynArray): variant; overload;

Create a TJSONVariant TJSONVariant array from a supplied array of values

function JSONVariant(const JSON: string): variant; overload;

Create a TJSONVariant instance from a given JSON content
- typical usage may be:
  var doc: variant;
  json: string;
  begin
    doc := JSONVariant("{"test":1234,"name":"John\n\r"}");
    assert(doc.test=1234); // access via late binding
    assert(doc.name="John"#13);
    assert(doc.name2=null); // unknown properties returns null
    json := doc; // to convert a TJSONVariant to JSON, just assign to a string
    assert(json="{"test":1234,"name":"John\n\r"}");
  end;

- note that FPC does not allow to set values by late-binding

function JSONVariantData(const JSONVariant: variant): PJSONVariantData;

Access to a TJSONVariant instance members
- e.g. Kind, Count, Names[] or Values[]
- will raise an exception if the supplied variant is not a TJSONVariant
- this function is safer than TJSONVariant(JSONVariant)

function JSONVariantDataSafe(const JSONVariant: variant; ExpectedKind: TJSONVariantKind=jvUndefined): PJSONVariantData;

Access to a TJSONVariant instance members
- e.g. Kind, Count, Names[] or Values[]
- will return a read-only fake TJSONVariant with Kind=jvUndefined if the supplied variant is not a TJSONVariant
- if ExpectedKind is jvArray of jvObject, it would return a fake TJSONVariant with Kind=jvUndefined if the JSONVariant kind does not match - so you can write:
  var _a: integer;
  _arr: PJSONVariantData;
...  
  _arr := JSONVariantDataSafe(_variant,jvArray);
  SetLength(result,_arr.Count);
  for _a := 0 to _arr.Count-1 do
    result[_a] := _arr.Values[_a];

in the above code, _arr.Count will be 0 if _variant.Kind<>jvArray
- this function is safer than TJSONVariant(JSONVariant)

function JSONVariantFromConst(const constValues: array of variant): variant;

Create a TJSONVariant TJSONVariant array from a supplied array of values
function NowToIso8601: string;
Compute the ISO-8601 JSON text representation of the current date/time value
- e.g. "2015-06-27T20:59:29"

function ObjectToJSON(Instance: TObject; StoreClassName: boolean=false): string;
Compute the JSON representation of an object published properties
- handle only simple types of properties, not nested class instances
- any TList/TObjectList/TCollection will be serialized as JSON array

procedure RegisterClassForJSON(const Classes: array of TClass);
Register the class types to be created from its name
- used e.g. by JSONToNewObject() or TJSONVariantData.ToNewObject

function RTTIPropInfoTypeName(PropInfo: TRTTIPropInfo): string;
Return a string corresponding to the type name, as stored in the RTTI
- e.g. 'TDateTime', 'TByteDynArray', 'TModTime', 'TCreateTime'

procedure SetInstanceProp(Instance: TObject; PropInfo: TRTTIPropInfo; const Value: variant);
Set the value of a published property from a variant

function ShortStringToString(Buffer: PByteArray): string;
Convert ASCII-7 latin text, encoded as a shortstring buffer, into a string
- as such, this function would work with NEXTGEN encoded RTTI

function StartWithPropName(const PropName1, PropName2: string): boolean;
Check that two ASCII-7 latin text do match

function StringToJSON(const Text: string): string;
Compute the quoted JSON string corresponding to the supplied text

function UTF8FileToString(const aFileName: TFileName): string;
Read an UTF-8 (JSON) file into a native string
- file should be existing, otherwise an exception is raised

function ValueToJSON(const Value: variant): string;
Compute the JSON representation of a variant value
- will work for simple types, or TJSONVariant object or array

function VarRecToValue(const V: TVarRec; out wasString: boolean): string;
Convert an "array of const" parameter value into its string representation

Variables implemented in the SynCrossPlatformJSON unit

JSONVariantType: TInvokeableVariantType;
The custom variant type definition registered for TJSONVariant
27.49. SynCrossPlatformREST.pas unit

**Purpose**: Minimum stand-alone cross-platform REST process for mORMot client
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

### Units used in the *SynCrossPlatformREST* unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>SynCrossPlatformCrypto</em></td>
<td>Cryptographic cross-platform units</td>
<td>1859</td>
</tr>
<tr>
<td><em>SynCrossPlatformJSON</em></td>
<td>Minimum standand-alone cross-platform JSON process using variants</td>
<td>1861</td>
</tr>
<tr>
<td><em>SynCrossPlatformSpecific</em></td>
<td>System-specific cross-platform units</td>
<td>1892</td>
</tr>
</tbody>
</table>

#### SynCrossPlatformREST class hierarchy

![SynCrossPlatformREST class hierarchy diagram]

#### Objects implemented in the *SynCrossPlatformREST* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
### Objects

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERestException</td>
<td>Exception type raised when working with REST access</td>
<td>1871</td>
</tr>
<tr>
<td>EServiceException</td>
<td>Exception type raised when working with interface-based service process</td>
<td>1872</td>
</tr>
<tr>
<td>IServiceAbstract</td>
<td>All generated client interfaces will inherit from this abstract parent</td>
<td>1877</td>
</tr>
<tr>
<td>TServiceClientAbstract</td>
<td>Abstract ancestor to all client-side interface-based services</td>
<td>1877</td>
</tr>
<tr>
<td>TServiceClientAbstract</td>
<td>Abstract ancestor to all TClientDriven interface-based services</td>
<td>1878</td>
</tr>
<tr>
<td>TSQLAuthGroup</td>
<td>Table containing the available user access rights for authentication</td>
<td>1876</td>
</tr>
<tr>
<td>TSQLAuthUser</td>
<td>Table containing the Users registered for authentication</td>
<td>1876</td>
</tr>
<tr>
<td>TSQLModel</td>
<td>Store the database model</td>
<td>1874</td>
</tr>
<tr>
<td>TSQLModelInfo</td>
<td>Store information of one TSQLRecord class</td>
<td>1872</td>
</tr>
<tr>
<td>TSQLModelInfoPropInfo</td>
<td>Store information of one TSQLRecord published property</td>
<td>1872</td>
</tr>
<tr>
<td>TSQLRecord</td>
<td>Abstract ORM class to access remote tables</td>
<td>1874</td>
</tr>
<tr>
<td>TSQLRest</td>
<td>Abstract REST access class</td>
<td>1879</td>
</tr>
<tr>
<td>TSQLRestClientHTTP</td>
<td>REST client via HTTP</td>
<td>1885</td>
</tr>
<tr>
<td>TSQLRestClientURI</td>
<td>REST client access class</td>
<td>1883</td>
</tr>
<tr>
<td>TSQLRestLogClientThread</td>
<td>Thread used to asynchronously log to a remote client</td>
<td>1884</td>
</tr>
<tr>
<td>TSQLRestRoutingAbstract</td>
<td>Class used to determine the protocol of interface-based services</td>
<td>1878</td>
</tr>
<tr>
<td>TSQLRestRoutingJSON_RPC</td>
<td>JSON/RPC protocol for interface-based services</td>
<td>1879</td>
</tr>
<tr>
<td>TSQLRestRoutingREST</td>
<td>Default simple REST protocol for interface-based services</td>
<td>1878</td>
</tr>
<tr>
<td>TSQLRestServerAuthentication</td>
<td>Abstract class used for client authentication</td>
<td>1885</td>
</tr>
<tr>
<td>TSQLRestServerAuthenticationDefault</td>
<td>MORMot secure RESTful authentication scheme</td>
<td>1885</td>
</tr>
<tr>
<td>TSQLRestServerAuthenticationNone</td>
<td>MORMot weak RESTful authentication scheme</td>
<td>1885</td>
</tr>
<tr>
<td>TSQLTableJSON</td>
<td>Handle a JSON result table, as returned by MORMot's REST server ORM</td>
<td>1872</td>
</tr>
</tbody>
</table>

**ERestException** = `class(Exception)`

*Exception type raised when working with REST access*
**TSQLTableJSON** = **class**(TJSONTableObject)

Handle a JSON result table, as returned by mORMot's REST server ORM
- this class is expected to work with TSQLRecord instances only
- it will let any "RowID" JSON key match TSQLRecord.ID property

**function** FillOne(aValue: TSQLRecord; aSeekFirst: boolean=false): boolean;
To be called in a loop to iterate through all data rows
- if returned true, Object published properties will contain this row

**EServiceException** = **class**(ERestException)
Exception type raised when working with interface-based service process

**TSQLModelInfoPropInfo** = **class**(TObject)
Store information of one TSQLRecord published property

Kind: TSQLFieldKind;
The property field type

Name: **string**;
The name of the published property
- e.g. 'FirstName'

RTTI: TRTTIPropInfo;
RTTI information about the published property

TypeName: **string**;
The property type name, as retrieved from RTTI

**constructor** CreateFrom(aRTTI: TRTTIPropInfo);
Initialize the instance

**TSQLModelInfo** = **class**(TObject)
Store information of one TSQLRecord class

AllFields: TSQLFieldBits;
Specifies all fields, including simple and BLOB fields

BlobFields: TSQLFieldBits;
Specifies the BLOB fields

CreateTimeFields: TSQLFieldBits;
Specifies the TCreateTime fields

HasKind: TSQLFieldKinds;
Contains all published properties kind

HasTimeFields: boolean;
TRUE if has TModTime or TCreateTime fields
ModAndCreateTimeFields: TSQLFieldBits;
    Specifies the TModTime and TCreateTime fields

ModTimeFields: TSQLFieldBits;
    Specifies the TModTime fields

Name: string;
    The short name of the class
    - i.e. 'People' for TSQLRecordPeople

Prop: TSQLModelInfoPropInfoDynArray;
    Information about every published property
    - first is always the ID field

RecordFields: TSQLFieldBits;
    Specifies the Record fields

SimpleFields: TSQLFieldBits;
    Specifies the "simple" fields, i.e. all non BLOB fields

Table: TSQLRecordClass;
    The TSQLRecord class type itself

VariantFields: TSQLFieldBits;
    Specifies the Variant fields

constructor CreateFromRTTI(aTable: TSQLRecordClass);
    Initialize the class member for the supplied TSQLRecord

destructor Destroy; override;
    Finalize the memory used

function FieldBitsToFieldNames(const FieldBits: TSQLFieldBits): string;
    Return the corresponding field names

function FieldNamesToFieldBits(const FieldNames: string; IncludeModTimeFields: boolean): TSQLFieldBits;
    FieldNames='' to retrieve simple fields, '*' all fields, or as specified

function SQLSelect(const FieldNames: string): string;
    Compute the 'SELECT ... FROM ...' corresponding to the supplied fields

function ToJSON(Value: TSQLRecord; const Fields: TSQLFieldBits): string; overload;
    Save the specified record as JSON

function ToJSONAdd(Client: TSQLRest; Value: TSQLRecord; ForceID: boolean; const FieldNames: string): string;
    Save the specified record as JSON for record adding

function ToJSONUpdate(Client: TSQLRest; Value: TSQLRecord; const FieldNames: string; ForceID: boolean): string;
    Save the specified record as JSON for record update

procedure ComputeFieldsBeforeWrite(aClient: TSQLRest; Value: TSQLRecord; AndCreate: Boolean);
    Set TModTime and TCreateTime fields
TSQLModel = class(TObject)

Store the database model

constructor Create(const Tables: array of TSQLRecordClass; const aRoot: string = 'root');

Initialize the Database Model
- set the Tables to be associated with this Model, as TSQLRecord classes
- set the optional Root URI path of this Model - default is 'root'

destructor Destroy; override;

Finalize the memory used

function GetTableIndex(const aTableName: string): integer; overload;
    Get index of aTable in Tables[], returns -1 if not found

function GetTableIndex(aTable: TSQLRecordClass): integer; overload;
    Get index of aTable in Tables[], returns -1 if not found

function GetTableIndexExisting(aTable: TSQLRecordClass): integer;
    Get index of aTable in Tables[], raise an ERestException if not found

function InfoExisting(aTable: TSQLRecordClass): TSQLModelInfo;
    Get the RTTI information for the specified class or raise an ERestException

procedure Add(Table: TSQLRecordClass);
    Register a new Table class to this Model

property Info: TSQLModelInfoDynArray read fInfo;
    The RTTI information for each class

property Root: string read fRoot;
    The Root URI path of this Database Model

TSQLRecord = class(TPersistent)

Abstract ORM class to access remote tables
- in comparison to mORMot.pas TSQLRecord published fields, dynamic arrays shall be defined as variant (since SynCrossPlatformJSON do not serialize)
- inherit from TPersistent to have RTTI for its published properties (SmartMobileStudio does not allow {$M+} in the source)

constructor Create(aClient: TSQLRest; const FieldNames, SQLWhere: string; const BoundsSQLWhere: array of const); overload;
    This constructor loads a record from a REST instance
    - you can bind parameters by using ? in the SQLWhere clause
    - use DateTimeToSQL() for date/time database fields
    - FieldNames='' retrieve simple fields, '*' all fields, or as specified

constructor Create(aClient: TSQLRest; aID: TID; ForUpdate: boolean=false); overload;
    This constructor loads a record from a REST instance from its ID
constructor Create; overload; virtual;
    This constructor initializes the record

constructor CreateAndFillPrepare(aClient: TSQLRest; const FieldNames, SQLWhere: string; const BoundsSQLWhere: array of const);
    This constructor ask the server for a list of matching records
    - you can bind parameters by using ? in the SQLWhere clause
    - use DateTimeToSQL() for date/time database fields
    - FieldNames="" retrieve simple fields, '*' all fields, or as specified
    - then you can also loop through all rows with
    while Rec.FillOne do
        dosomethingwith(Rec);

constructor CreateFromVariant(const aValue: variant);
    This constructor will loads a record from its variant representation
    - will call internaly the FromJSON() method

destructor Destroy; override;
    Finalize the record memory

function FillOne: boolean;
    Fill all published properties of this object with the next available row of data, as returned by CreateAndFillPrepare() constructor

function FillRewind: boolean;
    Go to the first data row, as returned by CreateAndFillPrepare(), then fill all published properties of this object
    - you can use it e.g. as:
    while Rec.FillOne do
        dosomethingwith(Rec);
    if Rec.FillRewind then
        repeat
            dosomeotherthingwith(Rec);
        until not Rec.FillOne;

function FromJSON(const aJSON: string): boolean;
    Fill the specified record from the supplied JSON

function FromVariant(const aValue: variant): boolean;
    Fill the specified record from its variant representation

function RecordClass: TSQLRecordClass;
    Return the class type of this TSQLRecord

function ToJSON(aModel: TSQLModel; aFieldNames: string=''): string;
    Get the object properties as JSON
    - FieldNames="" to retrieve simple fields, '*' all fields, or as specified

function ToVariant: variant;
    Get the object properties as a TJSONVariant document

property FillTable: TSQLTableJSON read FFill;
    Contains the TSQLTableJSON instance after CreateAndFillPrepare()
property ID: TID read fID write fID;
   Stores the record's primary key

property InternalState: cardinal read fInternalState;
   Internal state counter of the mORMot server at last access time
   - can be used to check if retrieved data may be out of date

TSQLAuthGroup = class(TSQLRecord)
   Table containing the available user access rights for authentication
   - added here since should be part of the model
   - no wrapper is available to handle AccessRights, since for security reasons it is not available remotely from client side

property AccessRights: string read fAccessRights write fAccessRights;
   A textual representation of a TSQLAccessRights buffer

property Ident: string read fIdent write fIdent stored AS_UNIQUE;
   The access right identifier, ready to be displayed
   - the same identifier can be used only once (this column is marked as unique via a "stored AS_UNIQUE" (i.e. "stored false") attribute)

property SessionTimeout: integer read fSessionTimeOut write fSessionTimeOut;
   The number of minutes a session is kept alive

TSQLAuthUser = class(TSQLRecord)
   Table containing the Users registered for authentication

property Data: TSQLRawBlob read fData write fData;
   Some custom data, associated to the User
   - Server application may store here custom data
   - its content is not used by the framework but 'may' be used by your application

property DisplayName: string read fDisplayName write fDisplayName;
   The User Name, as may be displayed or printed

property GroupRights: TID read fGroup write fGroup;
   The associated access rights of this user in TSQLAuthGroup
   - access rights are managed by group
   - note that 'Group' field name is not allowed by SQLite

property LogonName: string read fLogonName write fLogonName stored AS_UNIQUE;
   The User identification Name, as entered at log-in
   - the same identifier can be used only once (this column is marked as unique via a "stored AS_UNIQUE" - i.e. "stored false" - attribute), and therefore indexed in the database (e.g. hashed in TSQLRestStorageInMemory)

property PasswordHashHexa: string read fPasswordHashHexa write fPasswordHashHexa;
   The hexa encoded associated SHA-256 hash of the password
property PasswordPlain: string write SetPasswordPlain;

   Able to set the PasswordHashHexa field from a plain password content
- in fact, PasswordHashHexa := SHA256('salt'+PasswordPlain) in UTF-8

TServiceClientAbstract = class(TInterfacedObject)

Abstract ancestor to all client-side interface-based services
- any overridden class will in fact call the server to execute its methods
- inherited classes are in fact the main entry point for all interface-based services, without any
interface use:
   aCalculator := TServiceCalculator.Create(aClient);
   try
      aIntegerResult := aCalculator.Add(10,20);
   finally
      aCalculator.Free;
   end;
- under SmartMobileStudio, calling Free is mandatory only for sicClientDriven mode (to release
   the server-side associated session), so e.g. for a sicShared instance, you can safely write:
   aIntegerResult := TServiceCalculator.Create(aClient).Add(10,20);
- as you already noted, server-side interface-based services are in fact consummed without any
interface in this cross-platform unit!

constructor Create(aClient: TSQLRestClientURI); virtual;
   Initialize the fake instance
- this method will synchronously (i.e. blocking) check the server contract according to the one
expected by the client
- overridden constructors will set the parameters expected by the server

property Client: TSQLRestClientURI read GetClient;
   The associated TSQLRestClientURI instance

property ContractExpected: string read GetContractExpected;
   The published service contract, as expected by both client and server

property InstanceImplementation: TServiceInstanceImplementation read GetInstanceImplementation;
   How this instance lifetime is expected to be handled

property ServiceName: string read GetServiceName;
   The unmangled remote service name

property ServiceURI: string read GetServiceURI;
   The URI to access to the remote service

IServiceAbstract = interface(IInterface)

All generated client interfaces will inherit from this abstract parent

property Client: TSQLRestClientURI read GetClient;
   The associated TSQLRestClientURI instance

property ContractExpected: string read GetContractExpected;
   The published service contract, as expected by both client and server
property InstanceImplementation: TServiceInstanceImplementation read GetInstanceImplementation;

How this instance lifetime is expected to be handled

property RunningInstance: TServiceClientAbstract read GetRunningInstance;

The client class instance currently implementing this interface

property ServiceName: string read.GetServiceName;

The unmangled remote service name

property ServiceURI: string read.GetServiceURI;

The URI to access to the remote service

TServiceClientAbstractClientDriven = class(TServiceClientAbstract)

Abstract ancestor to allsicClientDriven interface-based services
- since server-side life-time is driven by the client, this kind of class expects an explicit call to aService.Free (even on SmartMobileStudio)

creator Create(aClient: TSQLRestClientURI); override;

Initialize the fake instance and create the remote per-client session
- raise an EServiceException if a per-client session was already started for the specified TSQLRestClientURI
- overriden constructors will set the parameters expected by the server

destructor Destroy; override;

This overriden method (called at aService.Free) will notify the server

property ClientID: string read fClientID;

The currently running instance ID on the server side
- only one instance is allowed per TSQLRestClientURI process

TSQLRestRoutingAbstract = class(TObject)

Class used to determine the protocol of interface-based services
- see TSQLRestRoutingREST and TSQLRestRoutingJSON_RPC for overridded methods - NEVER set this abstract TSQLRestRoutingAbstract class on TSQLRest.ServicesRouting property!

class procedure ClientSideInvoke(var uri: string; const method, params, clientDrivenID: string; var sent: string); virtual; abstract;

At Client Side, compute URI and BODY according to the routing scheme
- abstract implementation which is to be overriden
- as input, "method" should be the method name to be executed for "uri", "params" should contain the incoming parameters as JSON array (with []), and "clientDriven" ID should contain the optional Client ID value
- at output, should update the HTTP "uri" corresponding to the proper routing, and should return the corresponding HTTP body within "sent"

TSQLRestRoutingREST = class(TSQLRestRoutingAbstract)

Default simple REST protocol for interface-based services
- this is the default protocol used by TSQLRest
class procedure ClientSideInvoke(var uri: string; const method, params, clientDrivenID: string; var sent: string); override;

At Client Side, compute URI and BODY according to RESTful routing scheme
- e.g. on input uri='root/Calculator', method='Add', params='[1,2]' and clientDrivenID='1234' ->
on output uri='root/Calculator/Add/1234' and sent='[1,2]' 

TSQLRestRoutingJSON_RPC = class(TSQLRestRoutingAbstract)
JSON/RPC protocol for interface-based services
- alternative to the TSQLRestRoutingREST default protocol set by TSQLRest

class procedure ClientSideInvoke(var uri: string; const method, params, clientDrivenID: string; var sent: string); override;

At Client Side, compute URI and BODY according to JSON/RPC routing scheme
- e.g. on input uri='root/Calculator', method='Add', params='[1,2]' and clientDrivenID='1234' ->
on output uri='root/Calculator/Add/1234' and sent={"method":"Add","params":[1,2],"id":1234} 

TSQLRest = class(TObject)
Abstract REST access class

constructor Create(aModel: TSQLModel; aOwnModel: boolean=false); virtual;

Initialize the class, and associate it to a specified database Model
- if aOwnModel is TRUE, this class destructor will free aModel instance 

destructor Destroy; override;

Will release the associated Model, if aOwnModel was TRUE at Create() 

function Add(Value: TSQLRecord; SendData: boolean; ForceID: boolean=false; FieldNames: string=''): TID; virtual;

Create a new member, returning the newly created ID, or 0 on error
- if SendData is true, content of Value is sent to the server as JSON
- if ForceID is true, client sends the Value.ID field to use this ID for adding the record (instead of
  a database-generated ID)
- by default, only simple fields are pushed to the server, but you may specify a CSV list of field
  values to be transmitted - including blobs, which will be sent as base-64 encoded JSON

function BatchAdd(Value: TSQLRecord; SendData: boolean; ForceID: boolean=false; FieldNames: string=''): integer;

Create a new member in current BATCH sequence
- similar to Add(), but in BATCH mode: nothing is sent until BatchSend()
- returns the corresponding index in the current BATCH sequence, -1 on error
- you can set FieldNames='"' to sent simple fields, '*' to add all fields (including BLOBs), or specify
  a CSV list of added fields
- this method will always compute and send TCreateTime/TModTime fields 

function BatchCount: integer;

Retrieve the current number of pending transactions in the BATCH sequence
- every call to BatchAdd/Update/Delete methods increases this count
function BatchDelete(ID: TID): integer; overload;

Delete a member in current BATCH sequence
- similar to Delete(), but in BATCH mode: nothing is sent until BatchSend()
- returns the corresponding index in the current BATCH sequence, -1 on error
- deleted record class is the TSQLRecordClass used at BatchStart() call: it will fail if no class was specified for this BATCH sequence

function BatchDelete(Table: TSQLRecordClass; ID: TID): integer; overload;

Delete a member in current BATCH sequence
- similar to Delete(), but in BATCH mode: nothing is sent until BatchSend()
- returns the corresponding index in the current BATCH sequence, -1 on error
- with this overloaded method, the deleted record class is specified: no class shall have been set at BatchStart() call, or should be the same

function BatchDelete(Value: TSQLRecord): integer; overload;

Delete a member in current BATCH sequence
- similar to Delete(), but in BATCH mode: nothing is sent until BatchSend()
- returns the corresponding index in the current BATCH sequence, -1 on error

function BatchSend(var Results: TIDDynArray): integer;

Execute a BATCH sequence started by BatchStart() method
- send all pending BatchAdd/Update/Delete statements to the remote server
- will return the URI Status value, i.e. 200/HTTP_SUCCESS OK on success
- a dynamic array of 64 bit integers will be created in Results, containing all ROWDID created for each BatchAdd call, or 200 (=HTTP_SUCCESS) for all successfull BatchUpdate/BatchDelete, or 0 on error
- any error during server-side process MUST be checked against Results[] (the main URI Status is 200 if about communication success, and won't imply that all statements in the BATCH sequence were successful

function BatchStart(aTable: TSQLRecordClass; AutomaticTransactionPerRow: cardinal=10000; BatchOptions: TSQLRestBatchOptions=[]): boolean; virtual;

Begin a BATCH sequence to speed up huge database change
- then call BatchAdd(), BatchUpdate() or BatchDelete() methods with the proper class or instance of the
- at BatchSend call, all the sequence transactions will be sent at once
- at BatchAbort call, all operations will be aborted
- expect one TSQLRecordClass as parameter, which will be used for the whole sequence (in this case, you can't mix classes in the same BATCH sequence)
- if no TSQLRecordClass is supplied, the BATCH sequence will allow any kind of individual record in BatchAdd/BatchUpdate/BatchDelete
- return TRUE on success, FALSE if aTable is incorrect or a previous BATCH sequence was already initiated
- this method includes a AutomaticTransactionPerRow parameter, which will let all BATCH process be executed on the server side within an unique transaction grouped by the given number of rows
**function** BatchUpdate(Value: TSQLRecord; FieldNames: string=''): integer;

*Update a member in current BATCH sequence*
- similar to Update(), but in BATCH mode: nothing is sent until BatchSend()
- returns the corresponding index in the current BATCH sequence, -1 on error
- you can set FieldNames="" to sent simple fields, '*' to add all fields (including BLOBs), or specify a CSV list of added fields
- this method will always compute and send any TModTime fields

**function** Delete(Table: TSQLRecordClass; ID: TID): boolean; virtual; abstract;

*Delete a member*

**function** ExecuteList(const SQL: string): TSQLTableJSON; virtual; abstract;

*Execute directly a SQL statement, returning a list of data rows or nil*

**function** MultiFieldValues(Table: TSQLRecordClass; const FieldNames, SQLWhere: string): TSQLTableJSON; overload;

*Execute directly a SQL statement, expecting a list of results*
- return a result table on success, nil on failure
- FieldNames="" retrieve simple fields, '*' all fields, or as specified

**function** MultiFieldValues(Table: TSQLRecordClass; const FieldNames, SQLWhere: string; const BoundsSQLWhere: array of const; LimitFirstRow: Boolean=false): TSQLTableJSON; overload;

*Execute directly a SQL statement, expecting a list of results*
- return a result table on success, nil on failure
- you can bind parameters by using ? in the SQLWhere clause
- use DateTimeToSQL() for date/time database fields
- FieldNames="" retrieve simple fields, '*' all fields, or as specified

**function** Retrieve(const FieldNames, SQLWhere: string; const BoundsSQLWhere: array of const; Value: TSQLRecord): boolean; overload;

*Get a member from a where clause*
- you can bind parameters by using ? in the SQLWhere clause
- use DateTimeToSQL() for date/time database fields
- FieldNames="" retrieve simple fields, '*' all fields, or as specified

**function** Retrieve(aID: TID; Value: TSQLRecord; ForUpdate: boolean=false): boolean; overload; virtual; abstract;

*Get a member from its ID*
- return true on success, and fill all simple fields

**function** RetrieveBlob(Table: TSQLRecordClass; aID: TID; const BlobFieldName: string; out BlobData: TSQLRawBlob): boolean; virtual; abstract;

*Get a blob field content from its record ID and supplied blob field name*
- returns true on success, and the blob binary data

**function** RetrieveList(Table: TSQLRecordClass; const FieldNames, SQLWhere: string; const BoundsSQLWhere: array of const): TObjectList; overload;

*Execute directly a SQL statement, returning a list of TSQLRecord*
- you can bind parameters by using ? in the SQLWhere clause
- use DateTimeToSQL() for date/time database fields
- FieldNames="" retrieve simple fields, '*' all fields, or as specified
function Update(Value: TSQLRecord; FieldNames: string=''): boolean; virtual;

Update a member
- you can let default FieldNames='' to update simple fields, '*' to update all fields (including BLOBs), or specify a CSV list of updated fields

procedure BatchAbort;

Abort a BATCH sequence started by BatchStart() method
- in short, nothing is sent to the remote server, and sequence is voided

procedure Log(Level: TSynLogInfo; Instance: TObject); overload;

Call this method to add some information to the log at a specified level
- overloaded method which will log the corresponding class name and address
- the supplied log level will be checked against TSQLRest.LogLevel
- use LogToFile() or LogToRemoteServer() to set the OnLog callback

procedure Log(E: Exception); overload;

Call this method to add some information to the log for an Exception
- will log the Exception class name and message, if sllException is set

procedure Log(Level: TSynLogInfo; const Text: string; Instance: TObject=nil); overload;

Call this method to add some information to the log at a specified level
- the supplied log level will be checked against TSQLRest.LogLevel
- if Instance is set, it will log the corresponding class name and address
- will compute the text line in the very same format as TSynLog class
- use LogToFile() or LogToRemoteServer() to set the OnLog callback

procedure Log(Level: TSynLogInfo; const Fmt: string; const Args: array of const; Instance: TObject=nil); overload;

Call this method to add some information to the log at a specified level
- overloaded method which will call Format() to render the text
- here the Fmt layout is e.g. '%s %d %g', as standard Format(), and not the same as with SynCommons' FormatUTF8()
- the supplied log level will be checked against TSQLRest.LogLevel
- if Instance is set, it will log the corresponding class name and address
- use LogToFile() or LogToRemoteServer() to set the OnLog callback

procedure LogToFile(LogLevel: TSynLogInfos; const aFolderName: TFileName=''; const aFileName: TFileName='');

Start the logging process into a file
- if no directory is specified, will use the current one
- if no file name is specified, will compute a new one with the current time stamp, in the specified directory

procedure LogToRemoteServer(LogLevel: TSynLogInfos; const aServer: string; aPort: integer=8091; const aRoot: string='LogService');

Start the logging process into a remote log server
- the server could be for instance a LogView tool running in server mode

property InternalState: cardinal read fInternalState;

Internal state counter of the mORMot server at last access time
- can be used to check if retrieved data may be out of date
property LogLevel: TSynLogInfos read fLogLevel write fLogLevel;
   The set of log events which will be logged by Log() overloaded methods
   - set to [] by default, meaning that log is disabled

property Model: TSQLModel read fModel;
   The associated data model

property OnLog: TOnSQLRestLog read fOnLog write fOnLog;
   The callback to be executed by Log() overloaded methods
   - if none is set, the instance won't log anything

property ServerTimeStamp: TTimeLog read GetServerTimeStamp;
   The current Date and Time, as retrieved from the server at connection

property ServicesRouting: TSQLRestRoutingAbstractClass read fServicesRouting;
   The access protocol to be used for interface-based services
   - is set to TSQLRestRoutingREST by default
   - you can set TSQLRestRoutingJSON_RPC if the server expects this protocol

TSQLRestClientURI = class(TSQLRest)
   REST client access class

constructor Create(aModel: TSQLModel; aOwnModel: boolean=false); override;
   Initialize the class, and associate it to a specified database Model
   - if aOwnModel is TRUE, this class destructor will free aModel instance

destructor Destroy; override;
   Will call SessionClose

function CallBackGetResult(const aMethodName: string; const aNameValueParameters: array of const; aTable: TSQLRecordClass; aID: TID=0): string;
   Decode "result":... content as returned by CallBackGet()
   - if no Table is expected, set aTable=nil (we do not define nil as default parameter, since the SMS compiler is sometimes confused)

function Connect: boolean;
   Connect to the REST server, and retrieve its time stamp offset
   - under SMS, you should not use this blocking version, but the overloaded asynchronous method

function Delete(Table: TSQLRecordClass; ID: TID): boolean; override;
   Delete a member

function ExecuteList(const SQL: string): TSQLTableJSON; override;
   Execute directly a SQL statement, returning a list of rows or nil
   - we expect reUrlEncodedSQL to be defined in AllowRemoteExecute on server side, since we will encode the SQL at URL level, so that all HTTP client libraries will accept this layout (e.g. Indy or AJAX)

function Retrieve(aID: TID; Value: TSQLRecord; ForUpdate: boolean=false): boolean; overload; override;
   Get a member from its ID using URI()
function RetrieveBlob(Table: TSQLRecordClass; aID: TID; const BlobFieldName: string; out BlobData: TSQLRawBlob): boolean; 
Get a blob field content from its record ID and supplied blob field name
- returns true on success, and the blob binary data, as directly retrieved from the server via a dedicated HTTP GET request

function SetUser(aAuthenticationClass: TSQLRestServerAuthenticationClass; const aUserName, aPassword: string; aHashedPassword: Boolean=False): boolean;
Authenticate an User to the current connected Server
- using TSQLRestServerAuthenticationDefault or TSQLRestServerAuthenticationNone
- will set Authentication property on success

procedure CallBackGet(const aMethodName: string; const aNameValueParameters: array of const; var Call: TSQLRestURIParams; aTable: TSQLRecordClass; aID: TID=0);
Wrapper to the protected URI method to call a method on the server
- perform a ModelRoot/[TableName/[ID/]]MethodName RESTful GET request
- if no Table is expected, set aTable=nil (we do not define nil as default parameter, since the SMS compiler is sometimes confused)

procedure CallRemoteService(aCaller: TServiceClientAbstract; const aMethodName: string; aExpectedOutputParamsCount: integer; const aInputParams: array of variant; out res: TVariantDynArray; aReturnsCustomAnswer: boolean=false);
Execute a specified interface-based service method on the server
- this blocking method would raise an EServiceException on error
- you should not call it, but directly TServiceClient* methods

procedure SessionClose;
Close the session initiated with SetUser()
- will reset Authentication property to nil

procedure URI(var Call: TSQLRestURIParams); virtual;
Method calling the remote Server via a RESTful command
- calls the InternalURI abstract method
- this method will sign the url, if authentication is enabled

property Authentication: TSQLRestServerAuthentication read fAuthentication;
If not nil, point to the current authentication session running

property OnlyJSONRequests: boolean read fOnlyJSONRequests write fOnlyJSONRequests;
Set this property to TRUE if the server expects only APPLICATION/JSON
- applies only for AJAX clients (i.e. SmartMobileStudio platform)
- true will let any remote call be identified as "preflighted requests", so will send an OPTIONS method prior to any request: may be twice slower
- the default is false, as in TSQLHttpServer.OnlyJSONRequests

TSQLRestLogClientThread = class(TThread)
Thread used to asynchronously log to a remote client

constructor Create(Owner: TSQLRest; const aServer: string; aPort: integer; const aRoot: string);
Initialize the thread
**Synopspe mORMot Framework**  
**Software Architecture Design 1.18**  
**Date: September 16, 2020**

```pascal
destructor Destroy; override;
    Finalize the thread

procedure LogToRemoteServerText(const Text: string);
    Log one line of text

TSQLRestServerAuthentication = class(TObject)
    Abstract class used for client authentication

constructor Create(const aUserName, aPassword: string; aHashedPassword: Boolean=false);
    Initialize client authentication instance, i.e. the User associated instance

destructor Destroy; override;
    Finalize the instance

property SessionID: cardinal read fSessionID;
    Contains the session ID used for the authentication

property User: TSQLAuthUser read fUser;
    Read-only access to the logged user information
    - only LogonName and PasswordHashHexa are set here

TSQLRestServerAuthenticationDefault = class(TSQLRestServerAuthentication)
    MORMot secure RESTful authentication scheme

TSQLRestServerAuthenticationNone = class(TSQLRestServerAuthentication)
    MORMot weak RESTful authentication scheme

TSQLRestClientHTTP = class(TSQLRestClientURI)
    REST client via HTTP
    - note that this implementation is not thread-safe yet

constructor Create(const aServer: string; aPort: integer; aModel: TSQLModel;
    aOwnModel: boolean=false; aHttps: boolean=false; const aProxyName: string='';
    const aProxyByPass: string=''; aSendTimeout: Cardinal=30000; aReceiveTimeout:
    Cardinal=30000; aConnectionTimeOut: cardinal=30000); reintroduce; virtual;
    Access to a mORMot server via HTTP

destructor Destroy; override;
    Finalize the connection

procedure SetHttpBasicAuthHeaders(const aUserName, aPasswordClear: RawUTF8);
    Force the HTTP headers of any request to contain some HTTP BASIC authentication, without
    creating any remote session
    - here the password should be given as clear content
    - potential use case is to use a mORMot client through a HTTPS proxy
    - then you can use SetUser(TSQLRestServerAuthenticationDefault,...) to define any another
    "mORMot only" authentication

property Connection: TAbstractHttpConnection read fConnection;
    The associated connection, if active
```
property KeepAlive: Integer read fKeepAlive write fKeepAlive;
  The keep-alive timeout, in ms (20000 by default)

property Parameters: TSQLRestConnectionParams read fParameters;
  The connection parameters

Types implemented in the SynCrossPlatformREST unit

RawUTF8 = string;
  Alias to share the same string type between client and server

TCreateTime = type TTimeLog;
  Used to define a field which shall be set at record creation

TID = type Int64;
  The TSQLRecord primary key is a 64 bit integer

TIDDynArray = array of TID;
  A dynamic array of TSQLRecord primary keys
  - used e.g. for BATCH process

TModTime = type TTimeLog;
  Used to define a field which shall be set at each modification

TOnSQLRestLog = procedure(const Text: string) of object;
  Callback event used to write some text to a logging system
  - could be a local file (not for SMS apps), or a remote log server
  - the Text is already in the same format than the one generated by TSynLog

TServiceClientAbstractClass = class of TServiceClientAbstract;
  Class type used to identify an interface-based service
  - we do not rely on interfaces here, but simply on abstract classes

TServiceInstanceImplementation = ( sicSingle, sicShared, sicClientDriven,
  sicPerSession, sicPerUser, sicPerGroup, sicPerThread);
  The possible Server-side instance implementation patterns for interface-based services
  - each interface-based service will be implemented by a corresponding class instance on the server:
    this parameter is used to define how class instances are created and managed
  - on the Client-side, each instance will be handled depending on the server side implementation (i.e.
    with sicClientDriven behavior if necessary)

TSQLAuthGroupClass = class of TSQLAuthGroup;
  Class of the table containing the available user access rights for authentication

TSQLFieldBit = 0..MAX_SQLFIELDDS-1;
  Used to identify the a field in a Table as in TSQLFieldBits

TSQLFieldBits = set of TSQLFieldBit;
  Used to store bit set for all available fields in a Table
  - in this unit, field at index [0] indicates TSQLRecord.ID

TSQLFieldKind = ( sftUnspecified, sftDateTime, sftTimeLog, sftBlob, sftModTime,
  sftCreateTime, sftRecord, sftVariant );
  A published property kind
  - does not match mORMot.pas TSQLFieldType: here we recognize only types which may expect a
special behavior in this unit

```pascal
TSQLFieldKinds = set of TSQLFieldKind;
A set of published property Kind

TSQLModelInfoDynArray = array of TSQLModelInfo;
Store information of several TSQLRecord class

TSQLModelInfoPropInfoDynArray = array of TSQLModelInfoPropInfo;
Store information of all TSQLRecord published properties

TSQLRawBlob = TByteDynArray;
Alias to share the same blob type between client and server

TSQLRestBatchOption = ( boInsertOrIgnore );
The available options for TSQLRest.BatchStart() process
- boInsertOrIgnore will create 'INSERT OR IGNORE' statements instead of plain 'INSERT' - by now, only direct SQLite3 engine supports it on server

TSQLRestBatchOptions = set of TSQLRestBatchOption;
A set of options for TSQLRest.BatchStart() process

TSQLRestRoutingAbstractClass = class of TSQLRestRoutingAbstract;
Used to define the protocol of interface-based services

TSQLRestServerAuthenticationClass = class of TSQLRestServerAuthentication;
Class used for client authentication

TSynLogInfo = ( sllNone, sllInfo, sllDebug, sllTrace, sllWarning, sllError, sllEnter, sllLeave, sllLastError, sllException, sllExceptionOS, sllMemory, sllStackTrace, sllFail, sllSQL, sllCache, sllResult, sllDB, sllHTTP, sllClient, sllServer, sllServiceCall, sllServiceReturn, sllUserAuth, sllCustom1, sllCustom2, sllCustom3, sllCustom4, sllNewRun, sllDDEError, sllDDDInfo );
The available logging events, as handled by our Cross-Platform units
- defined with the same values in SynCommons.pas
- sllInfo will log general information events
- sllDebug will log detailed debugging information
- sllTrace will log low-level step by step debugging information
- sllWarning will log unexpected values (not an error)
- sllError will log errors
- sllEnter will log every method start
- sllLeave will log every method exit
- sllLastError will log the GetLastError OS message
- sllException will log all exception raised - available since Windows XP
- sllExceptionOS will log all OS low-level exceptions (EDivByZero, ERangeError, EAccessViolation...)
- sllMemory will log memory statistics
- sllStackTrace will log caller's stack trace (it's by default part of TSynLogFamily.LevelStackTrace like sllError, sllException, sllExceptionOS, sllLastError and sllFail)
- sllFail was defined for TSynTestsLogged.Failed method, and can be used to log some customer-side assertions (may be notifications, not errors)
- sllSQL is dedicated to trace the SQL statements
- sllCache should be used to trace the internal caching mechanism
- sllResult could trace the SQL results, JSON encoded
- sllDB is dedicated to trace low-level database engine features
```
sllHTTP could be used to trace HTTP process
sllClient/sllServer could be used to trace some Client or Server process
sllServiceCall/sllServiceReturn to trace some remote service or library
sllUserAuth to trace user authentication (e.g. for individual requests)
sllCustom* items can be used for any purpose
sllNewRun will be written when a process opens a rotated log
sllDDDError will log any DDD-related low-level error information
sllDDDInfo will log any DDD-related low-level debugging information

```pascal
tsynLogInfos = set of tsynLogInfo;
```

*Used to define a set of logging level abilities
- i.e. a combination of none or several logging event
- e.g. use LOG_VERBOSE constant to log all events, or LOG_STACKTRACE to log all errors and exceptions

```pascal
TTimeLog = type Int64;
```

*Fast bit-encoded date and time value*

**Constants implemented in the SynCrossPlatformREST unit**

- **AS_UNIQUE** = false;
  - Used as "stored AS_UNIQUE" published property definition in TSQLRecord
- **ID_SQLFIELD**: TSQLFieldBit = TSQLFieldBit(0);
  - The first field in TSQLFieldBits is always ID/RowID
- **JSON_SQLDATE_MAGIC** = #$ef#$bf#$b1;
  - \"uFFF1 special code to mark ISO-8601 SQLDATE in JSON
  - e.g. \""\"uFFF12012-05-04\"" pattern
  - Unicode special char U+FFF1 is UTF-8 encoded as EF BF B1 bytes
  - as generated by DateTimeToSQL/TimeLogToSQL functions, and expected by our mORMot server
  - should be used with BoundsSQLWhere parameters, e.g. with FormatBind()

- **MAX_SQLFIELDS** = 256;
  - *Maximum number of fields in a database Table*

**Functions or procedures implemented in the SynCrossPlatformREST unit**

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlobToVariant</td>
<td>Convert a binary blob into its base-64 representation</td>
<td>1889</td>
</tr>
<tr>
<td>DateTimeToSQL</td>
<td>Convert a date/time to a ISO-8601 string format for SQL '?' inlined parameters</td>
<td>1889</td>
</tr>
<tr>
<td>DateTimeToTTimeLog</td>
<td>Compute a TTimeLog value from Delphi date/time type</td>
<td>1889</td>
</tr>
<tr>
<td>FormatBind</td>
<td>Can be used to create a statement with inlined parameters</td>
<td>1889</td>
</tr>
<tr>
<td>GetOutHeader</td>
<td>Retrieve one header from a low-level HTTP response</td>
<td>1890</td>
</tr>
<tr>
<td>GUIDToVariant</td>
<td>Convert a TGUID instance into a string value</td>
<td>1890</td>
</tr>
<tr>
<td>HttpBodyToVariant</td>
<td>Convert a THttpBody binary content into a variant value</td>
<td>1890</td>
</tr>
</tbody>
</table>
### Synopse mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsRowID</td>
<td>True if PropName is either 'ID' or 'RowID'</td>
<td>1890</td>
</tr>
<tr>
<td>ObjectToVariant</td>
<td>Convert any TSQLRecord class instance into a TJSONVariant type</td>
<td>1890</td>
</tr>
<tr>
<td>SHA256Compute</td>
<td>Hash the supplied text values after UTF-8 encoding</td>
<td>1890</td>
</tr>
<tr>
<td>TimeLogToSQL</td>
<td>Convert a TTimeLog value into a ISO-8601 string format for SQL '?' inlined parameters</td>
<td>1890</td>
</tr>
<tr>
<td>ToDigit2</td>
<td>Returns a string with 2 digits</td>
<td>1890</td>
</tr>
<tr>
<td>ToDigit4</td>
<td>Returns a string with 4 digits</td>
<td>1890</td>
</tr>
<tr>
<td>TimeLogToDateT</td>
<td>Convert a TTimeLog value into the Delphi date/time type</td>
<td>1890</td>
</tr>
<tr>
<td>TimeLogToIso8601</td>
<td>Convert a TTimeLog value into an ISO-8601 encoded date/time text</td>
<td>1890</td>
</tr>
<tr>
<td>UrlDecode</td>
<td>Decode a text as defined by RFC 3986</td>
<td>1890</td>
</tr>
<tr>
<td>UrlEncode</td>
<td>Encode name=value pairs as defined by RFC 3986</td>
<td>1890</td>
</tr>
<tr>
<td>VariantToBlob</td>
<td>Convert a base-64 encoded blob into its binary representation</td>
<td>1890</td>
</tr>
<tr>
<td>VariantToEnum</td>
<td>Convert a text or integer enumeration representation into its ordinal value</td>
<td>1891</td>
</tr>
<tr>
<td>VariantToGUID</td>
<td>Convert a string value into a TGUID instance</td>
<td>1891</td>
</tr>
<tr>
<td>VariantToHttpBody</td>
<td>Convert a variant value into a THttpBody binary</td>
<td>1891</td>
</tr>
</tbody>
</table>

**function** BlobToVariant(const Blob: TSQLRawBlob): variant;

*Convert a binary blob into its base-64 representation*

**function** DateTimeToSQL(DT: TDateTime): string;

*Convert a date/time to a ISO-8601 string format for SQL '?' inlined parameters*
- if DT=0, returns ""
- if DT contains only a date, returns the date encoded as '\uFFF1YYYY-MM-DD'
- if DT contains only a time, returns the time encoded as '\uFFF1Thh:mm:ss'
- otherwise, returns the ISO-8601 date and time encoded as '\uFFF1YYYY-MM-DDThh:mm:ss'
- to be used e.g. as in:
  aRec.CreateAndFillPrepare(Client, 'Datum<?', [DateTimeToSQL(Now)]);
- see TimeLogToSQL() if you are using TTimeLog/TModTime/TCreateTime values

**function** DateTimeToTTimeLog(Value: TDateTime): TTimeLog;

*Compute a TTimeLog value from Delphi date/time type*

**function** FormatBind(const SQLWhere: string; const BoundsSQLWhere: array of const): string;

*Can be used to create a statement with inlined parameters*
- use DateTimeToSQL() for date/time database fields
function GetOutHeader(const Call: TSQLRestURIPrams; const Name: string): string;
    Retrieve one header from a low-level HTTP response
    - use e.g. location := GetOutHeader(Call,'location');

function GUIDToVariant(const GUID: TGUID): variant;
    Convert a TGUID instance into a string value

function HttpBodyToVariant(const HttpBody: THttpBody): variant;
    Convert a THttpBody binary content into a variant value
    - will use a variant of type string as mean of proprietary raw binary storage: you need to use
      VariantToHttpBody() to get the value back from the variant

function IsRowID(const PropName: string): boolean;
    True if PropName is either 'ID' or 'RowID'

function ObjectToVariant(value: TSQLRecord): variant;
    Convert any TSQLRecord class instance into a TJSONVariant type

function SHA256Compute(const Values: array of string): string;
    Hash the supplied text values after UTF-8 encoding
    - as expected by the framework server

function TimeLogToSQL(const TimeStamp: TTimeLog): string;
    Convert a TTimeLog value into a ISO-8601 string format for SQL '?' inlined parameters
    - follows the same pattern as DateToSQL or DateTimeToSQL functions, i.e. will return the date or
      time encoded as '１YYYY-MM-DDThh:mm:ss'

function ToDigit2(value: integer): string;
    Returns a string with 2 digits
    - the supplied value should be in 0..99 range

function ToDigit4(value: integer): string;
    Returns a string with 4 digits
    - the supplied value should be in 0..9999 range

function TTimeLogToDateTime(Value: TTimeLog): TDateTime;
    Convert a TTimeLog value into the Delphi date/time type

function TTimeLogToIso8601(Value: TTimeLog): string;
    Convert a TTimeLog value into an ISO-8601 encoded date/time text

function UrlDecode(const aValue: string): string;
    Decode a text as defined by RFC 3986

function UrlEncode(const aValue: string): string; overload;
    Encode a text as defined by RFC 3986

function UrlEncode(const aNameValueParameters: array of const): string; overload;
    Encode name=value pairs as defined by RFC 3986

function VariantToBlob(const Value: variant): TSQLRawBlob;
    Convert a base-64 encoded blob into its binary representation
function VariantToEnum(const Value: variant; const TextValues: array of string): integer;
    Convert a text or integer enumeration representation into its ordinal value

function VariantToGUID(const value: variant): TGUID;
    Convert a string value into a TGUID instance

function VariantToHttpBody(const value: variant): THttpBody;
    Convert a variant value into a THttpBody binary
    - will use a variant of type string as mean of proprietary raw binary storage: format is limited to HttpBodyToVariant() conversion

Variables implemented in the SynCrossPlatformREST unit

    The text equivalency of each logging level, as written in the log content
    - and expected by TSynLog and our LogView tool

LOG_STACKTRACE: TSynLogInfos;
    Contains the logging levels for which stack trace should be dumped
    - which are mainly exceptions or application errors

LOG_VERBOSE: TSynLogInfos;
    Can be set to TSQLRest.LogLevel in order to log all available events

NO_SQLFIELDBITS: TSQLFieldBits;
    Contains no field bit set
27.50. SynCrossPlatformSpecific.pas unit

*Purpose:* System-specific cross-platform units
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynCrossPlatformSpecific unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCrtSock</td>
<td>Classes implementing TCP/UDP/HTTP client and server protocol</td>
<td>1083</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

**Objects implemented in the SynCrossPlatformSpecific unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAbstractHttpConnection</td>
<td>Abstract class for HTTP client connection</td>
<td>1893</td>
</tr>
<tr>
<td>TMutex</td>
<td>Cross-platform thread safe locking</td>
<td>1892</td>
</tr>
<tr>
<td>TSQLRestConnectionParams</td>
<td>The connection parameters, as stored and used by TAbstractHttpConnection</td>
<td>1893</td>
</tr>
<tr>
<td>TSQLRestURIPrams</td>
<td>Used to store the request of a REST call</td>
<td>1892</td>
</tr>
</tbody>
</table>

**TMutex = class(TObject)**

*Cross-platform thread safe locking*
- will use TMonitor on the newest Delphi platforms

**TSQLRestURIPrams = object(TObject)**

*Used to store the request of a REST call*

**InBody: THttpBody;**

*Input parameter containing the caller message body*

**InHead: string;**

*Input parameter containing the caller message headers*
OutBody: THttpBody;
   Output parameter to be set to the response message body

OutHead: string;
   Output parameter to be set to the response message header

OutStatus: cardinal;
   Output parameter to be set to the HTTP status integer code

Url: string;
   Input parameter containing the caller URI

UrlWithoutSignature: string;
   Caller URI, without any appended signature

Verb: string;
   Input parameter containing the caller method

function OutBodyUtf8: string;
   Get the response message body as UTF-8

procedure Init(const aUrl,aVerb,aUTF8Body: string);
   Set the caller content

TSQLRestConnectionParams = record
   The connection parameters, as stored and used by TAbstractHttpConnection

   CONNECTIONTIMEOUT: integer;
      The connection timeout, in ms

   HTTPS: boolean;
      If the connection should be HTTPS

   PORT: integer;
      The server port

   ProxyByPass: string;
      The optional proxy password to be used

   ProxyName: string;
      The optional proxy name to be used

   RECEIVE_TIMEOUT: cardinal
      The timeout when receiving data, in ms

   SEND_TIMEOUT: cardinal;
      The timeout when sending data, in ms

   SERVER: string;
      The server name or IP address

TAbstractHttpConnection = class(TObject)
   Abstract class for HTTP client connection
constructor Create(const aParameters: TSQLRestConnectionParams); virtual;

This is the main entry point for all HTTP clients
- connect to http://aServer:aPort or https://aServer:aPort
- optional aProxyName may contain the name of the proxy server to use, and aProxyByPass an
  optional semicolon delimited list of host names or IP addresses, or both, that should not be
  routed through the proxy

procedure URI(var Call: TSQLRestURIParams; const InDataType: string; KeepAlive: integer); virtual; abstract;

Perform the request
- this is the main entry point of this class
- inherited classes should override this abstract method

property ActualConnection: TObject read fOpaqueConnection;

Opaque access to the effective connection class instance
- which may be a TFPHttpClient, a TIdHTTP or a TWinHttpAPI

property Parameters: TSQLRestConnectionParams read fParameters;

The connection parameters

property Server: string read fURL;

The remote server full URI
- e.g. 'http://myserver:888/

Types implemented in the SynCrossPlatformSpecific unit

TAbstractHttpConnectionClass = class of TAbstractHttpConnection;

Define the inherited class for HTTP client connection

THttpBody = array of byte;

Will store input and output HTTP body content
- HTTP body may not match the string type, and could be binary
- this kind of variable is compatible with NextGen version of the compiler

Constants implemented in the SynCrossPlatformSpecific unit

HTTP_ACCEPTED = 202;

HTTP Status Code for "Accepted"

HTTP_BADGATEWAY = 502;

HTTP Status Code for "Bad Gateway"

HTTP_BADREQUEST = 400;

HTTP Status Code for "Bad Request"

HTTP_CONTINUE = 100;

HTTP Status Code for "Continue"

HTTP_CREATED = 201;

HTTP Status Code for "Created"

HTTP_FORBIDDEN = 403;

HTTP Status Code for "Forbidden"

HTTP_FOUND = 302;
HTTP Status Code for "Found"

HTTP_GATEWAYTIMEOUT = 504;

HTTP Status Code for "Gateway Timeout"

HTTP_HTTPVERSIONNONSUPPORTED = 505;

HTTP Status Code for "HTTP Version Not Supported"

HTTP_MOVEDPERMANENTLY = 301;

HTTP Status Code for "Moved Permanently"

HTTP_MULTIPLECHOICES = 300;

HTTP Status Code for "Multiple Choices"

HTTP_NOCONTENT = 204;

HTTP Status Code for "No Content"

HTTP_NONAUTHORIZEDINFO = 203;

HTTP Status Code for "Non-Authoritative Information"

HTTP_NOTACCEPTABLE = 406;

HTTP Status Code for "Not Acceptable"

HTTP_NOTALLOWED = 405;

HTTP Status Code for "Method Not Allowed"

HTTP_NOTFOUND = 404;

HTTP Status Code for "Not Found"

HTTP_NOTIMPLEMENTED = 501;

HTTP Status Code for "Not Implemented"

HTTP_NOTMODIFIED = 304;

HTTP Status Code for "Not Modified"

HTTP_PARTIALCONTENT = 206;

HTTP Status Code for "Partial Content"

HTTP_PROXYAUTHREQUIRED = 407;

HTTP Status Code for "Proxy Authentication Required"

HTTP_SEEOTHER = 303;

HTTP Status Code for "See Other"

HTTP_SERVERERROR = 500;

HTTP Status Code for "Internal Server Error"

HTTP_SUCCESS = 200;

HTTP Status Code for "Success"

HTTP_SWITCHINGPROTOCOLS = 101;

HTTP Status Code for "Switching Protocols"

HTTP_TEMPORARYREDIRECT = 307;

HTTP Status Code for "Temporary Redirect"

HTTP_TIMEOUT = 408;
HTTP Status Code for "Request Time-out"

HTTP_UNAUTHORIZED = 401;

HTTP Status Code for "Unauthorized"

HTTP_UNAVAILABLE = 503;

HTTP Status Code for "Service Unavailable"

HTTP_USE_PROXY = 305;

HTTP Status Code for "Use Proxy"

JSON_CONTENT_TYPE = 'application/json; charset=UTF-8';

MIME content type used for JSON communication

Functions or procedures implemented in the SynCrossPlatformSpecific unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetNextCSV</td>
<td>Will return the next CSV value from the supplied text</td>
<td>1896</td>
</tr>
<tr>
<td>HttpBodyToText</td>
<td>Convert a UTF-8 binary buffer into texts</td>
<td>1896</td>
</tr>
<tr>
<td>HttpConnectionClass</td>
<td>Gives access to the class type to implement a HTTP connection</td>
<td>1896</td>
</tr>
<tr>
<td>TextToHttpBody</td>
<td>Convert a text into UTF-8 binary buffer</td>
<td>1896</td>
</tr>
</tbody>
</table>

function GetNextCSV(const str: string; var index: Integer; var res: string; Sep: char='''','; resultTrim: boolean=false): boolean;

Will return the next CSV value from the supplied text

procedure HttpBodyToText(const Body: THttpBody; var Text: string);

Convert a UTF-8 binary buffer into texts

function HttpConnectionClass: TAbstractHttpConnectionClass;

Gives access to the class type to implement a HTTP connection
- will use WinHTTP API (from our SynCrtSock) under Windows
- will use Indy for Delphi on other platforms
- will use fcl-web (fphttpclient) with FreePascal

function TextToHttpBody(const Text: string): THttpBody;

Convert a text into UTF-8 binary buffer
27.51. SynCrossPlatformTests.pas unit

*Purpose:* Regression tests for mORMot's cross-platform units
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

### Units used in the *SynCrossPlatformTests* unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCrossPlatformCrypto</td>
<td>Cryptographic cross-platform units</td>
<td>1859</td>
</tr>
<tr>
<td>SynCrossPlatformJSON</td>
<td>Minimum stand-alone cross-platform JSON process using variants</td>
<td>1861</td>
</tr>
<tr>
<td>SynCrossPlatformREST</td>
<td>Minimum stand-alone cross-platform REST process for mORMot client</td>
<td>1870</td>
</tr>
<tr>
<td>SynCrossPlatformSpecific</td>
<td>System-specific cross-platform units</td>
<td>1892</td>
</tr>
</tbody>
</table>

### Objects implemented in the *SynCrossPlatformTests* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSynCrossPlatformClient</td>
<td>Regression tests of our CrossPlatform units</td>
<td>1898</td>
</tr>
<tr>
<td>TSynCrossPlatformTests</td>
<td>Regression tests of our CrossPlatform units</td>
<td>1898</td>
</tr>
<tr>
<td>TSynTest</td>
<td>Generic class for performing simple tests</td>
<td>1897</td>
</tr>
</tbody>
</table>

TSynTest = class(TObject)

*Generic class for performing simple tests*
- purpose of this ancestor is to have RTTI for its published methods, which will contain the tests

*Failed: cardinal;*

*How many Check() call did failed*
Ident: string;
The test case name

Passed: cardinal;
How many Check() call did pass

Tests: TPublishedMethodDynArray;
The registered tests, i.e. all published methods of this class

constructor Create(const aIdent: string='');
Create the test instance
- this constructor will add all published methods to the internal test list, accessible via the
  Count/TestName/TestMethod properties

procedure Check(test: Boolean; const Msg: string=''); overload;
Validate a test

procedure Run(LogToConsole: boolean);
Run all tests

TSynCrossPlatformTests = class(TSynTest)
Regression tests of our CrossPlatform units

TSynCrossPlatformClient = class(TSynTest)
Regression tests of our CrossPlatform units

Types implemented in the SynCrossPlatformTests unit

TSynTestEvent = procedure of object;
As generated by mORMotWrappers.pas ! the prototype of an individual test
- to be used with TSynTest descendants
27.52. mORMot.pas unit

**Purpose:** Common ORM and SOA classes for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The mORMot unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.1.1</td>
<td>Client-Server framework</td>
<td>2543</td>
</tr>
<tr>
<td>DI-2.1.1.1</td>
<td>RESTful framework</td>
<td>2543</td>
</tr>
<tr>
<td>DI-2.1.1.2.1</td>
<td>In-Process communication</td>
<td>2544</td>
</tr>
<tr>
<td>DI-2.1.1.2.2</td>
<td>Named Pipe protocol</td>
<td>2544</td>
</tr>
<tr>
<td>DI-2.1.1.2.3</td>
<td>Windows Messages protocol</td>
<td>2544</td>
</tr>
<tr>
<td>DI-2.1.1.2.4</td>
<td>HTTP/1.1 protocol</td>
<td>2545</td>
</tr>
<tr>
<td>DI-2.1.2</td>
<td>UTF-8 JSON format shall be used to communicate</td>
<td>2545</td>
</tr>
<tr>
<td>DI-2.1.3</td>
<td>The framework shall use an innovative ORM (Object-relational mapping) approach, based on classes RTTI (Runtime Type Information)</td>
<td>2546</td>
</tr>
<tr>
<td>DI-2.1.5</td>
<td>The framework shall offer a complete SOA process</td>
<td>2547</td>
</tr>
<tr>
<td>DI-2.2.1</td>
<td>The SQLite3 engine shall be embedded to the framework</td>
<td>2548</td>
</tr>
</tbody>
</table>

Units used in the mORMot unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
<tr>
<td></td>
<td>- implements AES,XOR,ADLER32,MDS,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynSSPI</td>
<td>Low level access to Windows SSPI/SChannel API for the Win32/Win64 platform</td>
<td>1714</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>Unit Name</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>SynSSPIAuth</td>
<td>Low level access to Windows Authentication for the Win32/Win64 platform</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under</td>
<td>1718</td>
</tr>
<tr>
<td></td>
<td>a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/cache/search/multithread/OS features</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTests</td>
<td>Unit test functions used by Synopse projects</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td>1832</td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynZip</td>
<td>Low-level access to ZLib compression (1.2.5 engine version)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under</td>
<td>1845</td>
</tr>
<tr>
<td></td>
<td>a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>
### mORMot class hierarchy

**Objects implemented in the mORMot unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBusinessLayerException</td>
<td>Exception raised in case of an error in project implementation logic</td>
<td>1948</td>
</tr>
<tr>
<td>ECommunicationException</td>
<td>Exception raised in case of a Client-Server communication error</td>
<td>1948</td>
</tr>
<tr>
<td>EInterfaceFactoryException</td>
<td>Exception dedicated to interface factory, e.g. services and mock/stubs</td>
<td>1948</td>
</tr>
<tr>
<td>EInterfaceResolverException</td>
<td>Exception raised in case of Dependency Injection (aka IoC) issue</td>
<td>1948</td>
</tr>
<tr>
<td>EModelError</td>
<td>Exception raised in case of wrong Model definition</td>
<td>1948</td>
</tr>
<tr>
<td>EORMBatchException</td>
<td>Exception raised in case of TSQLRestBatch problem</td>
<td>1948</td>
</tr>
<tr>
<td>EORMException</td>
<td>Generic parent class of all custom Exception types of this unit</td>
<td>1948</td>
</tr>
<tr>
<td>EParsingException</td>
<td>Exception raised in case of unexpected parsing error</td>
<td>1948</td>
</tr>
<tr>
<td>ESecurityException</td>
<td>Exception raised in case of any authentication error</td>
<td>1948</td>
</tr>
<tr>
<td>EServiceException</td>
<td>Exception dedicated to interface based service implementation</td>
<td>1948</td>
</tr>
<tr>
<td>ESQLTableException</td>
<td>Exception raised in case of incorrect TSQLTable.Step / Field*() use</td>
<td>1948</td>
</tr>
<tr>
<td>IAutoCreateFieldsResolve</td>
<td>Used to set the published properties of a TInjectableAutoCreateFields</td>
<td>2064</td>
</tr>
<tr>
<td>IMultiCallbackRedirect</td>
<td>Prototype of a class implementing redirection of a given interface</td>
<td>2101</td>
</tr>
<tr>
<td>IServiceRecordVersion</td>
<td>Service definition for master/slave replication notifications subscribe</td>
<td>2095</td>
</tr>
<tr>
<td>IServiceRecordVersionCallback</td>
<td>A callback interface used to notify a TSQLRecord modification in real time</td>
<td>2095</td>
</tr>
<tr>
<td>IServiceWithCallbackReleased</td>
<td>Service definition with a method which will be called when a callback interface instance is released on the client side</td>
<td>2096</td>
</tr>
<tr>
<td>PropWrap</td>
<td>Used to map a TPropInfo.GetProc/SetProc and retrieve its kind</td>
<td>1928</td>
</tr>
<tr>
<td>RecordRef</td>
<td>Useful object to type cast TRecordReference type value into explicit TSQLRecordClass and ID</td>
<td>2038</td>
</tr>
<tr>
<td>TAuthSession</td>
<td>Class used to maintain in-memory sessions</td>
<td>2140</td>
</tr>
<tr>
<td>TBlockingCallback</td>
<td>Asynchronous callback to emulate a synchronous/blocking process</td>
<td>2099</td>
</tr>
<tr>
<td>TClassInstance</td>
<td>Store information about a class, able to easily create new instances</td>
<td>1914</td>
</tr>
<tr>
<td>TClassProp</td>
<td>A wrapper to published properties of a class</td>
<td>1915</td>
</tr>
<tr>
<td>TClassType</td>
<td>A wrapper to class type information, as defined by the Delphi RTTI</td>
<td>1915</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TCollectionItemAutoCreateFields</td>
<td>Abstract TCollectionItem class, which will instantiate all its nested class</td>
<td>1942</td>
</tr>
<tr>
<td></td>
<td>published properties, then release them (and any T*ObjArray) when freed</td>
<td></td>
</tr>
<tr>
<td>TEnumType</td>
<td>A wrapper to enumeration type information, as defined by the Delphi RTTI</td>
<td>1916</td>
</tr>
<tr>
<td>TINIWriter</td>
<td>Simple writer to a Stream, specialized for writing an object as INI</td>
<td>1938</td>
</tr>
<tr>
<td>TInjectableAutoCreateFields</td>
<td>Abstract class which will auto-inject its dependencies (DI/IoC), and also</td>
<td>2064</td>
</tr>
<tr>
<td></td>
<td>manage the instances of its TPersistent/TSynPersistent published properties</td>
<td></td>
</tr>
<tr>
<td>TInjectableObject</td>
<td>Any service implementation class could inherit from this class to allow</td>
<td>2062</td>
</tr>
<tr>
<td></td>
<td>dependency injection aka SOLID DI/IoC by the framework</td>
<td></td>
</tr>
<tr>
<td>TInjectableObjectRest</td>
<td>Service implementation class, with direct access on the associated TS</td>
<td>2063</td>
</tr>
<tr>
<td></td>
<td>ServiceFactoryServer/TSQLRestServer instances</td>
<td></td>
</tr>
<tr>
<td>TInterfacedCallback</td>
<td>TInterfacedObject class which will notify a REST server when it is released</td>
<td>2099</td>
</tr>
<tr>
<td>TInterfacedCollection</td>
<td>Any TCollection used between client and server shall inherit from this class</td>
<td>1941</td>
</tr>
<tr>
<td>TInterfacedObjectAutoCreateFields</td>
<td>Abstract TInterfacedObject class, which will instantiate all its nested</td>
<td>1944</td>
</tr>
<tr>
<td></td>
<td>TPersistent/TSynPersistent published properties, then release them when</td>
<td></td>
</tr>
<tr>
<td></td>
<td>freed</td>
<td></td>
</tr>
<tr>
<td>TInterfacedObjectFromFactory</td>
<td>Abstract class handling a generic interface implementation class</td>
<td>2067</td>
</tr>
<tr>
<td>TInterfaceFactory</td>
<td>Class handling interface RTTI and fake implementation class</td>
<td>2064</td>
</tr>
<tr>
<td>TInterfaceFactoryGenerated</td>
<td>Class handling interface implementation generated from source</td>
<td>2068</td>
</tr>
<tr>
<td>TInterfaceFactoryRTTI</td>
<td>Class handling interface RTTI and fake implementation class</td>
<td>2067</td>
</tr>
<tr>
<td>TInterfaceMock</td>
<td>Used to mock an interface implementation via expect-run-verify pattern</td>
<td>2079</td>
</tr>
<tr>
<td>TInterfaceMockSpy</td>
<td>Used to mock an interface implementation via run-verify pattern</td>
<td>2080</td>
</tr>
<tr>
<td>TInterfaceResolver</td>
<td>Abstract factory class allowing to call interface resolution in cascade</td>
<td>2059</td>
</tr>
<tr>
<td>TInterfaceResolverForSingleInterface</td>
<td>Abstract factory class targeting a single kind of interface</td>
<td>2059</td>
</tr>
<tr>
<td>TInterfaceResolverInjected</td>
<td>Abstract factory class targeting any kind of interface</td>
<td>2060</td>
</tr>
<tr>
<td>TInterfaceStub</td>
<td>Used to stub an interface implementation</td>
<td>2073</td>
</tr>
<tr>
<td>TInterfaceStubLog</td>
<td>Used to keep track of one stubbed method call</td>
<td>2072</td>
</tr>
<tr>
<td>TInterfaceStubRule</td>
<td>Define a mocking / stubing rule used internally by TInterfaceStub</td>
<td>2071</td>
</tr>
<tr>
<td>TInterfaceStubRules</td>
<td>Define the rules for a given method as used internally by TInterfaceStub</td>
<td>2072</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TInterfaceTypeData</td>
<td>A wrapper to interface type information, as defined by the Delphi RTTI</td>
<td>1918</td>
</tr>
<tr>
<td>TJSONObjectDecoder</td>
<td>JSON object decoding and SQL generation, in the context of ORM process</td>
<td>1912</td>
</tr>
<tr>
<td>TJSONObjectSerializer</td>
<td>Simple writer to a Stream, specialized for writing an object as JSON</td>
<td>1938</td>
</tr>
<tr>
<td>TMethodInfo</td>
<td>A wrapper around a method definition</td>
<td>1928</td>
</tr>
<tr>
<td>TObjectVariant</td>
<td>A custom variant type used to have direct access to object published properties</td>
<td>2020</td>
</tr>
<tr>
<td>TOnInterfaceStubExecuteParamsAbstract</td>
<td>Abstract parameters used by TInterfaceStub.Executes() events callbacks</td>
<td>2068</td>
</tr>
<tr>
<td>TOnInterfaceStubExecuteParamsJSON</td>
<td>Parameters used by TInterfaceStub.Executes() events callbacks as JSON</td>
<td>2070</td>
</tr>
<tr>
<td>TOnInterfaceStubExecuteParamsVariant</td>
<td>Parameters used by TInterfaceStub.Executes() events callbacks as Variant</td>
<td>2069</td>
</tr>
<tr>
<td>TParamInfo</td>
<td>A wrapper around an individual method parameter definition</td>
<td>1927</td>
</tr>
<tr>
<td>TPersistentAutoCreateFields</td>
<td>Abstract TPersistent class, which will instantiate all its nested TPersistent class published properties, then release them (and any T*ObjArray) when freed</td>
<td>1942</td>
</tr>
<tr>
<td>TPropInfo</td>
<td>A wrapper containing a RTTI property definition</td>
<td>1920</td>
</tr>
<tr>
<td>TRawUTF8ObjectCache</td>
<td>Maintain information cache for a given key</td>
<td>1945</td>
</tr>
<tr>
<td>TRawUTF8ObjectCacheList</td>
<td>Manage a list of information cache, identified by a hashed key</td>
<td>1945</td>
</tr>
<tr>
<td>TRawUTF8ObjectCacheSettings</td>
<td>Used by TRawUTF8ObjectCacheList to manage a list of information cache</td>
<td>1944</td>
</tr>
<tr>
<td>TReturnInfo</td>
<td>A wrapper around method returned result definition</td>
<td>1927</td>
</tr>
<tr>
<td>TServiceContainer</td>
<td>A global services provider class</td>
<td>2093</td>
</tr>
<tr>
<td>TServiceContainerClient</td>
<td>A services provider class to be used on the client side</td>
<td>2098</td>
</tr>
<tr>
<td>TServiceContainerInterface</td>
<td>Used to lookup one service in a global list of interface-based services</td>
<td>2092</td>
</tr>
<tr>
<td>TServiceContainerInterfaceMethod</td>
<td>Used to lookup one method in a global list of interface-based services</td>
<td>2092</td>
</tr>
<tr>
<td>TServiceContainerServer</td>
<td>A services provider class to be used on the server side</td>
<td>2096</td>
</tr>
<tr>
<td>TServiceCustomAnswer</td>
<td>A record type to be used as result for a function method for custom content for interface-based services</td>
<td>2059</td>
</tr>
<tr>
<td>TServiceFactory</td>
<td>An abstract service provider, as registered in TServiceContainer</td>
<td>2082</td>
</tr>
<tr>
<td>TServiceFactoryClient</td>
<td>A service provider implemented on the client side</td>
<td>2090</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TServiceFactoryExecution</td>
<td>Internal per-method list of execution context as hold in TServiceFactory</td>
<td>1962</td>
</tr>
<tr>
<td>TServiceFactoryServer</td>
<td>A service provider implemented on the server side</td>
<td>2084</td>
</tr>
<tr>
<td>TServiceFactoryServerInstance</td>
<td>Server-side service provider uses this to store one internal instance</td>
<td>2084</td>
</tr>
<tr>
<td>TServiceMethod</td>
<td>Describe an interface-based service provider method</td>
<td>2053</td>
</tr>
<tr>
<td>TServiceMethodArgument</td>
<td>Describe a service provider method argument</td>
<td>2050</td>
</tr>
<tr>
<td>TServiceMethodExecute</td>
<td>Execute a method of a TInterfacedObject instance, from/to JSON</td>
<td>2057</td>
</tr>
<tr>
<td>TServiceRecordVersion</td>
<td>This class implements a service, which may be called to push notifications for master/slave replication</td>
<td>2098</td>
</tr>
<tr>
<td>TServiceRecordVersionCallback</td>
<td>This class implements a callback interface, able to write all remote ORM notifications to the local DB</td>
<td>2100</td>
</tr>
<tr>
<td>TServiceRunningContext</td>
<td>Will identify the currently running service on the server side</td>
<td>1961</td>
</tr>
<tr>
<td>TServicesPublishedInterfaces</td>
<td>Used to publish all Services supported by a TSQLRestServer instance</td>
<td>2157</td>
</tr>
<tr>
<td>TServicesPublishedInterfacesList</td>
<td>Used e.g. by TSQLRestServer to store a list of TServicesPublishedInterfaces</td>
<td>2157</td>
</tr>
<tr>
<td>TSQLAccessRights</td>
<td>Set the User Access Rights, for each Table</td>
<td>1962</td>
</tr>
<tr>
<td>TSQLAuthUser</td>
<td>Table containing the available user access rights for authentication</td>
<td>2138</td>
</tr>
<tr>
<td>TSQLAuthUser</td>
<td>Table containing the Users registered for authentication</td>
<td>2139</td>
</tr>
<tr>
<td>TSQLHttpPostServerDefinition</td>
<td>Parameters supplied to publish a TSQLRestServer via HTTP</td>
<td>2148</td>
</tr>
<tr>
<td>TSQLLocks</td>
<td>Used to store the locked record list, in a specified table</td>
<td>2022</td>
</tr>
<tr>
<td>TSQLModel</td>
<td>A Database Model (in a MVC-driven way), for storing some tables types as TSQLRecord classes</td>
<td>2032</td>
</tr>
<tr>
<td>TSQLModelRecordProperties</td>
<td>ORM properties associated to a TSQLRecord within a given model</td>
<td>2030</td>
</tr>
<tr>
<td>TSQLModelRecordPropertiesSQL</td>
<td>Pre-computed SQL statements for ORM operations for a given TSQLModelRecordProperties instance</td>
<td>2025</td>
</tr>
<tr>
<td>TSQLModelRecordReference</td>
<td>How a TSQLModel stores a foreign link to be cascaded</td>
<td>2031</td>
</tr>
<tr>
<td>TSQLMonitorUsage</td>
<td>ORM table used to store TSynMonitorUsage information in TSynMonitorUsageRest</td>
<td>2155</td>
</tr>
<tr>
<td>TSQLPropInfo</td>
<td>Abstract parent class to store information about a published property</td>
<td>1928</td>
</tr>
<tr>
<td>TSQLPropInfoCustom</td>
<td>Abstract information about a record-like property defined directly in code</td>
<td>1934</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>TSQLPropInfoCustomJSO N</td>
<td>Information about a custom property defined directly in code</td>
<td>1936</td>
</tr>
<tr>
<td>TSQLPropInfoList</td>
<td>Handle a read-only list of fields information for published properties</td>
<td>1937</td>
</tr>
<tr>
<td>TSQLPropInfoRecordFixedSize</td>
<td>Information about a fixed-size record property defined directly in code</td>
<td>1935</td>
</tr>
<tr>
<td>TSQLPropInfoRecordRTTI</td>
<td>Information about a record property defined directly in code</td>
<td>1935</td>
</tr>
<tr>
<td>TSQLPropInfoRecordTypEd</td>
<td>Information about a record property defined directly in code using RTTI</td>
<td>1934</td>
</tr>
<tr>
<td>TSQLPropInfoRTTI</td>
<td>Parent information about a published property retrieved from RTTI</td>
<td>1931</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIAnsi</td>
<td>Information about a AnsiString published property</td>
<td>1933</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIChar</td>
<td>Information about a character published property</td>
<td>1932</td>
</tr>
<tr>
<td>TSQLPropInfoRTTICurrency</td>
<td>Information about a fixed-decimal Currency published property</td>
<td>1933</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIDateTime</td>
<td>Information about a TDateTime published property</td>
<td>1933</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIDouble</td>
<td>Information about a floating-point Double published property</td>
<td>1933</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIDynArray</td>
<td>Information about a dynamic array published property</td>
<td>1933</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIEnum</td>
<td>Information about an enumeration published property</td>
<td>1932</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIID</td>
<td>Information about a TSQLRecord class TSQLRecord property</td>
<td>1950</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIInstance</td>
<td>Information about a TSQLRecord class property</td>
<td>1949</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIInt32</td>
<td>Information about an ordinal Int32 published property</td>
<td>1932</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIInt64</td>
<td>Information about an ordinal Int64 published property</td>
<td>1932</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIMany</td>
<td>Information about a TSQLRecord class TSQLRecordMany property</td>
<td>1951</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIObject</td>
<td>Information about a TSQLRecord class TStrings/TRawUTF8List/TCollection property</td>
<td>1950</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIRawBlob</td>
<td>Information about a TSQLRawBlob published property</td>
<td>1933</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIRawUnicode</td>
<td>Information about a RawUnicode published property</td>
<td>1933</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIRawUTF8</td>
<td>Information about a RawUTF8 published property</td>
<td>1933</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIRecordReference</td>
<td>Information about a TRecordReference/TRecordReferenceToBeDeleted published property</td>
<td>1949</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIRecordVersion</td>
<td>Information about a TRecordVersion published property</td>
<td>1950</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TSQLPropInfoRTTISet</td>
<td>Information about a set published property</td>
<td>1932</td>
</tr>
<tr>
<td>TSQLPropInfoRTTITID</td>
<td>Information about a TID published property</td>
<td>1949</td>
</tr>
<tr>
<td>TSQLPropInfoRTTITimeLog</td>
<td>Information about a TTimeLog published property</td>
<td>1933</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIUnixMSTime</td>
<td>Information about a TUnixMSTime published property</td>
<td>1933</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIUnixTime</td>
<td>Information about a TUnixTime published property</td>
<td>1933</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIVariant</td>
<td>Information about a variant published property</td>
<td>1934</td>
</tr>
<tr>
<td>TSQLPropInfoRTTIWide</td>
<td>Information about a WideString published property</td>
<td>1933</td>
</tr>
<tr>
<td>TSQLQueryCustom</td>
<td>Store one custom query parameters</td>
<td>2023</td>
</tr>
<tr>
<td>TSQLRecord</td>
<td>Root class for defining and mapping database records</td>
<td>1981</td>
</tr>
<tr>
<td>TSQLRecordCaseSensitive</td>
<td>Root class for defining and mapping database records with case-sensitive BINARY collation</td>
<td>2003</td>
</tr>
<tr>
<td>TSQLRecordCaseSensitiveExtended</td>
<td>Database records with BINARY collation and JSON_OPTIONS_FAST_EXTENDED variants</td>
<td>2003</td>
</tr>
<tr>
<td>TSQLRecordFill</td>
<td>Internal data used by TSQLRecord.FillPrepare()/FillPrepareMany() methods</td>
<td>1975</td>
</tr>
<tr>
<td>TSQLRecordFTS3</td>
<td>A base record, corresponding to a FTS3 table, i.e. implementing full-text</td>
<td>2042</td>
</tr>
<tr>
<td>TSQLRecordFTS3Porter</td>
<td>This base class will create a FTS3 table using the Porter Stemming algorithm</td>
<td>2043</td>
</tr>
<tr>
<td>TSQLRecordFTS3Unicode61</td>
<td>This base class will create a FTS3 table using the Unicode61 Stemming algorithm</td>
<td>2043</td>
</tr>
<tr>
<td>TSQLRecordFTS4</td>
<td>A base record, corresponding to a FTS4 table, which is an enhancement to FTS3</td>
<td>2043</td>
</tr>
<tr>
<td>TSQLRecordFTS4Porter</td>
<td>This base class will create a FTS4 table using the Porter Stemming algorithm</td>
<td>2043</td>
</tr>
<tr>
<td>TSQLRecordFTS4Unicode61</td>
<td>This base class will create a FTS4 table using the Unicode61 Stemming algorithm</td>
<td>2043</td>
</tr>
<tr>
<td>TSQLRecordFTS5</td>
<td>A base record, corresponding to a FTS5 table, which is an enhancement to FTS4</td>
<td>2044</td>
</tr>
<tr>
<td>TSQLRecordFTS5Porter</td>
<td>This base class will create a FTS5 table using the Porter Stemming algorithm</td>
<td>2044</td>
</tr>
<tr>
<td>TSQLRecordFTS5Unicode61</td>
<td>This base class will create a FTS5 table using the Unicode61 Stemming algorithm</td>
<td>2044</td>
</tr>
<tr>
<td>TSQLRecordHistory</td>
<td>Common ancestor for tracking changes on TSQLRecord tables</td>
<td>2150</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TSQLRecordInterfaced</td>
<td>Common ancestor for tables which should implement any interface</td>
<td>2050</td>
</tr>
<tr>
<td>TSQLRecordLog</td>
<td>A base record, with a JSON-logging capability</td>
<td>2048</td>
</tr>
<tr>
<td>TSQLRecordMany</td>
<td>Handle &quot;has many&quot; and &quot;has many through&quot; relationships</td>
<td>2045</td>
</tr>
<tr>
<td>TSQLRecordModification</td>
<td>Common ancestor for tracking TSQLRecord modifications</td>
<td>2149</td>
</tr>
<tr>
<td>TSQLRecordNoCase</td>
<td>Root class for defining and mapping database records with case-insensitive NOCASE collation</td>
<td>2003</td>
</tr>
<tr>
<td>TSQLRecordNoCaseExtended</td>
<td>Database records with NOCASE collation and JSON_OPTIONS_FAST_EXTENDED variants</td>
<td>2003</td>
</tr>
<tr>
<td>TSQLRecordNoCaseExtendedNoID</td>
<td>Database records with NOCASE collation and JSON_OPTIONS_FAST_EXTENDED variants, and itoNoIndex4TID option to avoid indexes on TID/T*ID properties</td>
<td>2003</td>
</tr>
<tr>
<td>TSQLRecordProperties</td>
<td>Some information about a given TSQLRecord class properties</td>
<td>1951</td>
</tr>
<tr>
<td>TSQLRecordPropertiesMapping</td>
<td>Allow custom field mapping of a TSQLRecord</td>
<td>2026</td>
</tr>
<tr>
<td>TSQLRecordRTree</td>
<td>A base record, corresponding to an R-Tree table of floating-point values</td>
<td>2040</td>
</tr>
<tr>
<td>TSQLRecordRTreeAbstract</td>
<td>An abstract base class, corresponding to an R-Tree table of values</td>
<td>2039</td>
</tr>
<tr>
<td>TSQLRecordRTreeInteger</td>
<td>A base record, corresponding to an R-Tree table of 32-bit integer values</td>
<td>2041</td>
</tr>
<tr>
<td>TSQLRecordServiceLog</td>
<td>Common ancestor for storing interface-based service execution statistics</td>
<td>2056</td>
</tr>
<tr>
<td>TSQLRecordServiceNotifications</td>
<td>Execution statistics used for DB-based asynchronous notifications</td>
<td>2056</td>
</tr>
<tr>
<td>TSQLRecordSigned</td>
<td>Common ancestor for tables with digitally signed RawUTF8 content</td>
<td>2049</td>
</tr>
<tr>
<td>TSQLRecordTableDeleted</td>
<td>ORM table used to store the deleted items of a versioned table</td>
<td>2152</td>
</tr>
<tr>
<td>TSQLRecordTimed</td>
<td>A base record, which will have creation and modification timestamp fields</td>
<td>2050</td>
</tr>
<tr>
<td>TSQLRecordVirtual</td>
<td>Parent of all virtual classes</td>
<td>2025</td>
</tr>
<tr>
<td>TSQLRecordVirtualTableAutoID</td>
<td>Record associated to Virtual Table implemented in Delphi, with ID generated automatically at INSERT</td>
<td>2220</td>
</tr>
<tr>
<td>TSQLRecordVirtualTableForcedID</td>
<td>Record associated to a Virtual Table implemented in Delphi, with ID forced at INSERT</td>
<td>2220</td>
</tr>
<tr>
<td>TSQLRest</td>
<td>A generic REpresentational State Transfer (REST) client/server class</td>
<td>2108</td>
</tr>
<tr>
<td>TSQLRestAcquireExecution</td>
<td>Used to store the execution parameters for a TSQLRest instance</td>
<td>2105</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TSQLRestBackgroundTimer</td>
<td>TThread able to run one or several tasks at a periodic pace, or do</td>
<td>2105</td>
</tr>
<tr>
<td></td>
<td>asynchronous interface or batch execution, with proper TSQLRest integration</td>
<td></td>
</tr>
<tr>
<td>TSQLRestBatch</td>
<td>Used to store a BATCH sequence of writing operations</td>
<td>1977</td>
</tr>
<tr>
<td>TSQLRestBatchLocked</td>
<td>Thread-safe class to store a BATCH sequence of writing operations</td>
<td>1980</td>
</tr>
<tr>
<td>TSQLRestCache</td>
<td>Implement a fast TSQLRecord cache, per ID, at the TSQLRest level</td>
<td>2103</td>
</tr>
<tr>
<td>TSQLRestCacheEntry</td>
<td>For TSQLRestCache, stores a table settings and values</td>
<td>2102</td>
</tr>
<tr>
<td>TSQLRestCacheEntryValue</td>
<td>For TSQLRestCache, stores a table values</td>
<td>2101</td>
</tr>
<tr>
<td>TSQLRestClient</td>
<td>A generic REpresentational State Transfer (REST) client</td>
<td>2193</td>
</tr>
<tr>
<td>TSQLRestClientCallbacks</td>
<td>Store information about registered interface callbacks</td>
<td>2195</td>
</tr>
<tr>
<td>TSQLRestClientRedirect</td>
<td>Rest client with redirection to another TSQLRest instance</td>
<td>2207</td>
</tr>
<tr>
<td>TSQLRestClientURI</td>
<td>A generic REpresentational State Transfer (REST) client with URI</td>
<td>2196</td>
</tr>
<tr>
<td>TSQLRestClientURIDll</td>
<td>Rest client with remote access to a server through a dll</td>
<td>2207</td>
</tr>
<tr>
<td>TSQLRestClientURIMessage</td>
<td>Rest client with remote access to a server through Windows messages</td>
<td>2208</td>
</tr>
<tr>
<td>TSQLRestClientURINamedPipe</td>
<td>Rest client with remote access to a server through a Named Pipe</td>
<td>2209</td>
</tr>
<tr>
<td>TSQLRestRoutingJSON_RPC</td>
<td>Calling context for a TSQLRestServerCallBack using JSON/RPC for</td>
<td>1975</td>
</tr>
<tr>
<td></td>
<td>interface-based services</td>
<td></td>
</tr>
<tr>
<td>TSQLRestRoutingREST</td>
<td>Calling context for a TSQLRestServerCallBack using simple REST for</td>
<td>1974</td>
</tr>
<tr>
<td></td>
<td>interface-based services</td>
<td></td>
</tr>
<tr>
<td>TSQLRestServer</td>
<td>A generic REpresentational State Transfer (REST) server</td>
<td>2159</td>
</tr>
<tr>
<td>TSQLRestServerAuthentication</td>
<td>Abstract class used to implement server-side authentication in TSQLRestServer</td>
<td>2142</td>
</tr>
<tr>
<td>TSQLRestServerAuthenticationDefault</td>
<td>MORMot secure RESTful authentication scheme</td>
<td>2144</td>
</tr>
<tr>
<td>TSQLRestServerAuthenticationHttpAbstract</td>
<td>Abstract class for implementing HTTP authentication</td>
<td>2146</td>
</tr>
<tr>
<td>TSQLRestServerAuthenticationHttpBasic</td>
<td>Authentication using HTTP Basic scheme</td>
<td>2147</td>
</tr>
<tr>
<td>TSQLRestServerAuthenticationNone</td>
<td>MORMot weak RESTful authentication scheme</td>
<td>2145</td>
</tr>
<tr>
<td>TSQLRestServerAuthenticationSignedURI</td>
<td>Secure authentication scheme using URL-level digital signature</td>
<td>2143</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TSQLRestServerAuthenticationSSPI</td>
<td>Authentication of the current logged user using Windows Security Support Provider Interface (SSPI) or GSSAPI library on Linux</td>
<td>2147</td>
</tr>
<tr>
<td>TSQLRestServerAuthenticationURI</td>
<td>Weak authentication scheme using URL-level parameter</td>
<td>2143</td>
</tr>
<tr>
<td>TSQLRestServerFullScreen</td>
<td>A REST server using only in-memory tables</td>
<td>2190</td>
</tr>
<tr>
<td>TSQLRestServerMethod</td>
<td>Description of a method-based service</td>
<td>1975</td>
</tr>
<tr>
<td>TSQLRestServerMonitor</td>
<td>Used for high-level statistics in TSQLRestServer.URI()</td>
<td>2154</td>
</tr>
<tr>
<td>TSQLRestServerNamedPipe</td>
<td>Server thread accepting connections from named pipes</td>
<td>2136</td>
</tr>
<tr>
<td>TSQLRestServerNamedPipeResponse</td>
<td>Server child thread dealing with a connection through a named pipe</td>
<td>2137</td>
</tr>
<tr>
<td>TSQLRestServerRemoteDB</td>
<td>A REST server using another TSQLRest instance for all its ORM process</td>
<td>2192</td>
</tr>
<tr>
<td>TSQLRestServerURI</td>
<td>Used to access a TSQLRestServer from its TSQLRestServerURIString URI</td>
<td>2156</td>
</tr>
<tr>
<td>TSQLRestServerURIContext</td>
<td>Abstract calling context for a TSQLRestServerCallBack event handler</td>
<td>1964</td>
</tr>
<tr>
<td>TSQLRestServerUriParameters</td>
<td>Structure used to specify custom request paging parameters for TSQLRestServer</td>
<td>2137</td>
</tr>
<tr>
<td>TSQLRestStorage</td>
<td>REST class with direct access to an external database engine</td>
<td>2179</td>
</tr>
<tr>
<td>TSQLRestStorageInMemory</td>
<td>REST storage with direct access to a TObjectList memory-stored table</td>
<td>2182</td>
</tr>
<tr>
<td>TSQLRestStorageInMemoryExternal</td>
<td>REST storage with direct access to a memory database, to be used as an external SQLite3 Virtual table</td>
<td>2188</td>
</tr>
<tr>
<td>TSQLRestStorageInMemoryUnique</td>
<td>Class able to handle a O(1) hashed-based search of a property</td>
<td>2182</td>
</tr>
<tr>
<td>TSQLRestStorageRecordBased</td>
<td>Abstract REST storage exposing some internal TSQLRecord-based methods</td>
<td>2181</td>
</tr>
<tr>
<td>TSQLRestStorageRemote</td>
<td>REST storage with redirection to another REST instance</td>
<td>2189</td>
</tr>
<tr>
<td>TSQLRestStorageShard</td>
<td>Abstract REST storage with redirection to several REST instances, implementing range ID partitioning for horizontal scaling</td>
<td>2189</td>
</tr>
<tr>
<td>TSQLRestTempStorage</td>
<td>Abstract class used for temporary in-memory storage of TSQLRecord</td>
<td>2152</td>
</tr>
<tr>
<td>TSQLRestTempStorageItem</td>
<td>Used to store an entry in the TSQLRestTempStorage class</td>
<td>2152</td>
</tr>
<tr>
<td>TSQLRestThread</td>
<td>A simple TThread for doing some process within the context of a REST instance</td>
<td>2135</td>
</tr>
<tr>
<td>TSQLRestURIParams</td>
<td>Store all parameters for a Client or Server method call</td>
<td>1960</td>
</tr>
<tr>
<td>TSQLRibbonTabParameters</td>
<td>Defines the settings for a Tab for User Interface generation</td>
<td>2024</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>TSQLTable</td>
<td>Wrapper to an ORM result table, statically stored as UTF-8 text</td>
<td>2004</td>
</tr>
<tr>
<td>TSQLTableFieldType</td>
<td>Store TSQLFieldtype and RTTI for a given TSQLTable field</td>
<td>2004</td>
</tr>
<tr>
<td>TSQLTableJSON</td>
<td>Store a read-only ORM result table from a JSON message</td>
<td>2020</td>
</tr>
<tr>
<td>TSQLTableRowVariant</td>
<td>A custom variant type used to have direct access to TSQLTable content</td>
<td>2019</td>
</tr>
<tr>
<td>TSQLTableRowVariantData</td>
<td>Memory structure used for our TSQLTableRowVariant custom variant type used to have direct access to TSQLTable content</td>
<td>2019</td>
</tr>
<tr>
<td>TSQLTableSortParams</td>
<td>Contains the parameters used for sorting</td>
<td>1912</td>
</tr>
<tr>
<td>TSQLTableWritable</td>
<td>Store a writable ORM result table, optionally read from a JSON message</td>
<td>2022</td>
</tr>
<tr>
<td>TSQLVirtualTable</td>
<td>Abstract class able to access a Virtual Table content</td>
<td>2213</td>
</tr>
<tr>
<td>TSQLVirtualTableBinary</td>
<td>A TSQLRestStorageInMemory-based virtual table using Binary storage</td>
<td>2219</td>
</tr>
<tr>
<td>TSQLVirtualTableCursor</td>
<td>Abstract class able to define a Virtual Table cursor</td>
<td>2216</td>
</tr>
<tr>
<td>TSQLVirtualTableCursorIndex</td>
<td>A generic Virtual Table cursor associated to Current/Max index properties</td>
<td>2216</td>
</tr>
<tr>
<td>TSQLVirtualTableCursorJSON</td>
<td>A Virtual Table cursor for reading a TSQLRestStorageInMemory content</td>
<td>2217</td>
</tr>
<tr>
<td>TSQLVirtualTableCursorLog</td>
<td>A Virtual Table cursor for reading a TSynLogFile content</td>
<td>2219</td>
</tr>
<tr>
<td>TSQLVirtualTableJSON</td>
<td>A TSQLRestStorageInMemory-based virtual table using JSON storage</td>
<td>2217</td>
</tr>
<tr>
<td>TSQLVirtualTableLog</td>
<td>Implements a read/only virtual table able to access a .log file, as created by TSynLog</td>
<td>2219</td>
</tr>
<tr>
<td>TSQLVirtualTableModule</td>
<td>Parent class able to define a Virtual Table module</td>
<td>2212</td>
</tr>
<tr>
<td>TSQLVirtualTablePrepared</td>
<td>The WHERE and ORDER BY statements as set by TSQLVirtualTable.Prepare</td>
<td>2211</td>
</tr>
<tr>
<td>TSQLVirtualTablePreparedConstraint</td>
<td>A WHERE constraint as set by the TSQLVirtualTable.Prepare() method</td>
<td>2210</td>
</tr>
<tr>
<td>TSQLVirtualTablePreparedOrderBy</td>
<td>An ORDER BY clause as set by the TSQLVirtualTable.Prepare() method</td>
<td>2211</td>
</tr>
<tr>
<td>TSynAuthenticationResult</td>
<td>TSynAuthentication* class using TSQLAuthUser/TSQLAuthGroup for credentials</td>
<td>2149</td>
</tr>
<tr>
<td>TSynAutoCreateFields</td>
<td>Our own empowered TPersistentAutoCreateFields-like parent class</td>
<td>1943</td>
</tr>
<tr>
<td>TSynAutoCreateFieldsLocked</td>
<td>Adding locking methods to a TSynAutoCreateFields with virtual constructor</td>
<td>1943</td>
</tr>
<tr>
<td>TSynJsonFileSettings</td>
<td>Abstract parent class able to store settings as JSON file</td>
<td>1944</td>
</tr>
</tbody>
</table>
**Objects** | **Description** | **Page**
---|---|---
TSynMonitorUsage | Abstract class to track, compute and store TSynMonitor detailed statistics | 1947
TSynMonitorUsageID | How the TSynMonitorUsage storage IDs are computed | 1946
TSynMonitorUsageRest | Will store TSynMonitorUsage information in TSQLMonitorUsage ORM tables | 2156
TSynValidateRest | Will define a validation to be applied to a TSQLRecord field, using if necessary an associated TSQLRest instance and a TSQLRecord class | 2209
TSynValidateUniqueField | Will define a validation for a TSQLRecord Unique text field | 2210
TSynValidateUniqueFields | Will define an unicity validation for a set of TSQLRecord text fields | 2210
TTypeInfo | A wrapper containing type information definition | 1919
TVirtualTableModuleProperties | Used to store and handle the main specifications of a TSQLVirtualTableModule | 2212

**TSQLTableSortParams** = record
Contains the parameters used for sorting
- FieldCount is 0 if was never sorted
- used to sort data again after a successful data update with TSQLTableJSON.FillFrom()

**TJSONObjectDecoder** = object(TObject)

JSON object decoding and SQL generation, in the context of ORM process
- this is the main process for marshalling JSON into SQL statements
- used e.g. by GetJSONObjectAsSQL() function or ExecuteFromJSON and InternalBatchStop methods

DecodedFieldNames: PRawUTF8Array;
Internal pointer over field names to be used after Decode() call
- either FieldNames, either Fields[] array as defined in Decode(), or external names as set by TSQLRestStorageExternal.JSONDecodedPrepareToSQL

DecodedFieldTypesToUnnest: PSQLDBFieldTypeArray;
Internal pointer over field types to be used after Decode() call
- to create 'INSERT INTO ... SELECT UNNEST(...)' or 'UPDATE ... FROM SELECT UNNEST(...)' statements for very efficient bulk writes in a PostgreSQL database
- as set by TSQLRestStorageExternal.JSONDecodedPrepareToSQL when cPostgreBulkArray flag is detected (for SynDBPostgres)

DecodedRowID: TID;
The ID=.. value as sent within the JSON object supplied to Decode()

FieldCount: integer;
Number of fields decoded in FieldNames[] and FieldValues[]
FieldNames: array[0..MAX_SQLFIELDS-1] of RawUTF8;

Contains the decoded field names

FieldTypeApproximation: array[0..MAX_SQLFIELDS-1] of TJSONObjectDecoderFieldType;

Decode() will set each field type approximation
- will recognize also JSON_BASE64_MAGIC/JSON_SQLDATE_MAGIC prefix

FieldValues: array[0..MAX_SQLFIELDS-1] of RawUTF8;

Contains the decoded field values

InlinedParams: TJSONObjectDecoderParams;

Set to TRUE if parameters are to be :(...): inlined

function EncodeAsSQL(Update: boolean): RawUTF8;

Encode as a SQL-ready INSERT or UPDATE statement
- after a successfull call to Decode()
- escape SQL strings, according to the official SQLite3 documentation (i.e. ' inside a string is stored as ')
- if InlinedParams was TRUE, it will create prepared parameters like 'COL1=(:"VAL1"),
  COL2=(:VAL2)'
- called by GetJSONObjectAsSQL() function or TSQLRestStorageExternal

function EncodeAsSQLPrepared(const TableName: RawUTF8; Occasion: TSQLOccasion; const UpdateIDFieldName: RawUTF8; BatchOptions: TSQLRestBatchOptions): RawUTF8;

Encode as a SQL-ready INSERT or UPDATE statement with ? as values
- after a successfull call to Decode()
- FieldValues[] content will be ignored
- Occasion can be only soInsert or soUpdate
- for soUpdate, will create UPDATE ... SET ... where UpdateIDFieldName=?
- you can specify some options, e.g. boInsertOrIgnore for solnsert

function FindFieldName(const FieldName: RawUTF8): integer;

Search for a field name in the current identified FieldNames[]

function SameFieldNames(const Fields: TRawUTF8DynArray): boolean;

Returns TRUE if the specified array match the decoded fields names
- after a successfull call to Decode()

procedure AddFieldValue(const FieldName,FieldValue: RawUTF8; FieldType: TJSONObjectDecoderFieldFieldtype);

Can be used after Decode() to add a new field in FieldNames/FieldValues
- so that EncodeAsSQL() will include this field in the generated SQL
- caller should ensure that the FieldName is not already defined in FieldNames[] (e.g. when the TRecordVersion field is forced)
- the caller should ensure that the supplied FieldValue will match the quoting/inlining expectations of Decode(TJSONObjectDecoderParams) - e.g. that string values are quoted if needed

procedure AssignFieldNamesTo(var Fields: TRawUTF8DynArray);

Set the specified array to the fields names
- after a successfull call to Decode()
procedure Decode(const JSON: RawUTF8; const Fields: TRawUTF8DynArray; Params: TJSONObjectDecoderParams; const RowID: TID=0; ReplaceRowIDWithID: Boolean=false); overload;

Decode the JSON object fields into FieldNames[] and FieldValues[]
- overloaded method expecting a RawUTF8 buffer, making a private copy of the JSON content to avoid unexpected in-place modification, then calling Decode(P: PUTF8Char) to perform the process

procedure Decode(var P: PUTF8Char; const Fields: TRawUTF8DynArray; Params: TJSONObjectDecoderParams; const RowID: TID=0; ReplaceRowIDWithID: Boolean=false); overload;

Decode the JSON object fields into FieldNames[] and FieldValues[]
- if Fields=nil, P should be a true JSON object, i.e. defined as "COL1"="VAL1" pairs, stopping at '}' or '['; otherwise, Fields[] contains column names and expects a JSON array as "VAL1","VAL2".. in P
- P should be after the initial '=' or '[' character, i.e. at first field
- P returns the next object start or nil on unexpected end of input
- P^ buffer will let the JSON be decoded in-place, so consider using the overloaded Decode(JSON: RawUTF8; ...) method
- FieldValues[] strings will be quoted and/or inlined depending on Params
- if RowID is set, a RowID column will be added within the returned content

procedure EncodeAsJSON(out result: RawUTF8);

Encode the FieldNames/FieldValues[] as a JSON object

TClassInstance = object(TObject)
Store information about a class, able to easily create new instances
- using this temporary storage will speed up the creation process
- any virtual constructor will be used, including for TCollection types

CollectionItemClass: TCollectionItemClass;
For TCollection instances, the associated TCollectionItem class

ItemClass: TClass;
The class type itself

ItemCreate: TClassInstanceItemCreate;
How the class instance is expected to be created

function CreateNew: TObject;
Create a new instance of the registered class

procedure Init(C: TClass);
Fill the internal information fields for a given class type

procedure SetCustomComment(var CustomComment: RawUTF8);
Compute the custom JSON commentary corresponding to this class
TClassProp = object(TObject)

A wrapper to published properties of a class
- start enumeration by getting a PClassProp with ClassProp()
- use PropCount, P := @PropList to get the first PPropInfo, and then P^.Next
- this enumeration is very fast and doesn't require any temporary memory, as in the TypInfo.GetPropInfos() PPropList usage
- for TSQLRecord, you should better use the RecordProps.Fields[] array, which is faster and contains the properties published in parent classes

PropCount: Word;
  Number of published properties in this object

PropList: record
  Point to a TPropInfo packed array
  - layout is as such, with variable TPropInfo storage size:
    PropList: array[1..PropCount] of TPropInfo
  - use TPropInfo.Next to get the next one:
    P := @PropList;
    for i := 1 to PropCount do begin
      // ... do something with P
      P := P^.Next;
    end;

function FieldProp(const PropName: shortstring): PPropInfo;
  Retrieve a Field property RTTI information from a Property Name

TClassType = object(TObject)

A wrapper to class type information, as defined by the Delphi RTTI

Used for DI-2.1.3 (page 2546).

ClassType: TClass;
  The class type

ParentInfo: PPTTypeInfo;
  The parent class type information

PropCount: SmallInt;
  The number of published properties

UnitName: string[255];
  The name (without .pas extension) of the unit were the class was defined
  - then the PClassProp follows: use the method ClassProp to retrieve its address

function ClassProp: PClassProp;
  Get the information about the published properties of this class
  - stored after UnitName memory
function InheritsFrom(AClass: TClass): boolean;
  
  *Fast and easy find if this class inherits from a specific class type*
  - you should rather consider using TTypeInfo.InheritsFrom directly

function RTTISize: integer;
  
  *Return the size (in bytes) of this class type information*
  - can be used to create class types at runtime

TEnumType = object(TObject)
  A wrapper to enumeration type information, as defined by the Delphi RTTI
  - we use this to store the enumeration values as integer, but easily provide a text equivalent,
  translated if necessary, from the enumeration type definition itself

*Used for DI-2.1.3 (page 2546).*

BaseType: PTypeInfo;
  *The base type of this enumeration*
  - always use PEnumType(typeinfo(TEnumType))^\.BaseType or more useful method
  PTypeInfo(typeinfo(TEnumType))^\.EnumBaseType before calling any of the methods below

MaxValue: Longint;
  *Same as ord(high(type)): not the enumeration count, but the highest index*

MinValue: Longint;
  *First value of enumeration type, typically 0*

NameList: string[255];
  *A concatenation of shortstrings, containing the enumeration names*
  - those shortstrings are not aligned whatsoever (even if FPC_REQUIRES_PROPER_ALIGNMENT is
  set)

OrdType: TOrdType;
  *Specify ordinal storage size and sign*
  - is preferred to MaxValue to identify the number of stored bytes

function GetCaption(const Value): string;
  *Get the corresponding caption name, without the first lowercase chars (otDone -> 'Done')*
  - return "string" type, i.e. UnicodeString for Delphi 2009+
  - internally call UnCamelCase() then System.LoadResStringTranslate() if available
  - Value will be converted to the matching ordinal value (byte or word)

function GetCaptionStrings(UsedValuesBits: Pointer=nil): string;
  *Get all caption names, ready to be display, as lines separated by #13#10*
  - return "string" type, i.e. UnicodeString for Delphi 2009+
  - if UsedValuesBits is not nil, only the corresponding bits set are added

function GetEnumName(const Value): PShortString;
  *Get the corresponding enumeration name*
  - return the first one if Value is invalid (>MaxValue)
  - Value will be converted to the matching ordinal value (byte or word)
function GetEnumNameAllAsJSONArray(TrimLeftLowerCase: boolean; UnCamelCased: boolean=false): RawUTF8;

Get all enumeration names as a JSON array of strings

function GetEnumNameOrd(Value: Integer): PShortString;

Get the corresponding enumeration name
- return the first one if Value is invalid (>MaxValue)

function GetEnumNameTrimed(const Value): RawUTF8;

Get the corresponding enumeration name, without the first lowercase chars ('Done' -> 'Done')
- Value will be converted to the matching ordinal value (byte or word)

function GetEnumNameTrimedValue(const EnumName: ShortString): Integer; overload;

Get the corresponding enumeration ordinal value, from its name without its first lowercase chars ('Done' will find otDone e.g.)
- return -1 if not found (don't use directly this value to avoid any GPF)

function GetEnumNameTrimedValue(Value: PUTF8Char; ValueLen: integer=0): Integer;

overload;

Get the corresponding enumeration ordinal value, from its name without its first lowercase chars ('Done' will find otDone e.g.)
- return -1 if not found (don't use directly this value to avoid any GPF)

function GetEnumNameValue(Value: PUTF8Char; ValueLen: integer; AlsoTrimLowerCase: boolean=true): Integer; overload;

Get the corresponding enumeration ordinal value, from its name
- if Value does start with lowercases 'a'..'z', they will be searched: e.g.
  GetEnumNameValue('sllWarning') will find sllWarning item
- if AlsoTrimLowerCase is TRUE, and EnumName does not start with lowercases 'a'..'z', they will be ignored: e.g. GetEnumNameValue('Warning') will find sllWarning item
- return -1 if not found (don't use directly this value to avoid any GPF)

function GetEnumNameValue(const EnumName: ShortString): Integer; overload;

Get the corresponding enumeration ordinal value, from its name
- if EnumName does start with lowercases 'a'..'z', they will be searched: e.g.
  GetEnumNameValue('sllWarning') will find sllWarning item
- if Value does not start with lowercases 'a'..'z', they will be ignored: e.g.
  GetEnumNameValue('Warning') will find sllWarning item
- return -1 if not found (don't use directly this value to avoid any GPF)

function GetEnumNameValue(Value: PUTF8Char): Integer; overload;

Get the corresponding enumeration ordinal value, from its name
- if Value does start with lowercases 'a'..'z', they will be searched: e.g.
  GetEnumNameValue('sllWarning') will find sllWarning item
- if Value does not start with lowercases 'a'..'z', they will be ignored: e.g.
  GetEnumNameValue('Warning') will find sllWarning item
- return -1 if not found (don't use directly this value to avoid any GPF)

functionSetNameAsDocVariant(Value: integer; FullSetsAsStar: boolean=false): variant;

Get the enumeration names corresponding to a set value, as a JSON array
function GetSetNameCSV(Value: integer; SepChar: AnsiChar=','; FullSetsAsStar: boolean=false): RawUTF8; overload;

Get the enumeration names corresponding to a set value

function SizeInStorageAsEnum: Integer;

Compute how many bytes this type will use to be stored as an enumerate

function SizeInStorageAsSet: Integer;

Compute how many bytes this type will use to be stored as a set

procedure AddCaptionStrings(Strings: TStrings; UsedValuesBits: Pointer=nil);

Add caption names, ready to be display, to a TStrings class
- add pointer(ord(element)) as Objects[] value
- if UsedValuesBits is not nil, only the corresponding bits set are added
- can be used e.g. to populate a combo box as such:
  PTypeInfo(TypeInfo(TMyEnum)).^(EnumBaseType^.AddCaptionStrings(ComboBox.Items);)

procedure GetEnumNameAll(var result: TRawUTF8DynArray; TrimLeftLowerCase: boolean); overload;

Retrieve all element names as a dynamic array of RawUTF8
- names could be optionally trimmed left from their initial lower chars

procedure GetEnumNameAll(var result: RawUTF8; const Prefix: RawUTF8=''; quotedValues: boolean=false; const Suffix: RawUTF8=''; trimedValues: boolean=false; unCamelCased: boolean=false); overload;

Retrieve all element names as CSV, with optional quotes

procedure GetEnumNameTrimedAll(var result: RawUTF8; const Prefix: RawUTF8=''; quotedValues: boolean=false; const Suffix: RawUTF8='');

Retrieve all trimed element names as CSV

procedure GetSetNameCSV(W: TTextWriter; Value: integer; SepChar: AnsiChar=','; FullSetsAsStar: boolean=false); overload;

Get the enumeration names corresponding to a set value

procedure SetEnumFromOrdinal(out Value; Ordinal: Integer);

Store an enumeration value from its ordinal representation

TInterfaceTypeData = record
A wrapper to interface type information, as defined by the Delphi RTTI

IntfFlags: TIntfFlags;

Ancestor
TTypeInfo = object(TObject)

A wrapper containing type information definition
- user types defined as an alias don't have this type information:
  type NewType = OldType;
- user types defined as new types have this type information:
  type NewType = type OldType;

Used for DI-2.1.3 (page 2546).

Kind: TTypeKind;
The value type family

Name: ShortString;
The declared name of the type ('String','Word','RawUnicode'...)

function AnsiStringCodePage: integer;
Recognize most used string types, returning their code page
- will recognize TSQLRawBlob as the fake CP_SQLRAWBLOB code page
- will return the exact code page since Delphi 2009, from RTTI
- for non Unicode versions of Delphi, will recognize WinAnsiString as CODEPAGE_US,
  RawUnicode as CP_UTF16, RawByteString as CP_RAWBYTESTRING, AnsiString as 0, and any
  other type as RawUTF8

function ClassCreate: TObject;
Create an instance of the corresponding class
- will call TObject.Create, or TSQLRecord.Create virtual constructor
- will raise EParsingException if class cannot be constructed on the fly, e.g. for a plain
  TCollectionItem class

function ClassFieldCount(onlyWithoutGetter: boolean): integer;
Get the number of published properties in this class
- you can count the plain fields without any getter function, if you do need only the published
  properties corresponding to some value actually stored, and ignore e.g. any textual conversion

function ClassSQLFieldType: TSQLFieldType;
Get the SQL type of this Delphi class type
- returns either sftObject, sftID, sftMany or sftUnknown

function ClassType: PClassType;
Get the class type information

function DynArrayItemSize: integer;
Get the dynamic array size (in bytes) of the stored item

function DynArrayItemType(aDataSize: PInteger= nil): PTypeInfo;
Get the dynamic array type information of the stored item

function DynArraySQLFieldType: TSQLFieldType;
Get the SQL type of the items of a dynamic array

function EnumBaseType: PEnumType;
Get the enumeration type information
function FloatType: TFloatType;
   For floating point types, get the storage size and precision

function GetSQLFieldType: TSQLErrorType;
   Get the SQL type of this Delphi type, as managed with the database driver

function InheritsFrom(AClass: TClass): boolean;
   Fast and easy find if a class type inherits from a specific class type

function InterfaceAncestor: PTypeInfo;
   Get the ancestor/parent of a given interface type information
   - returns nil if this type has no parent

function InterfaceGUID: PGUID;
   Get the TGUID of a given interface type information
   - returns nil if this type is not an interface

function InterfaceType: PInterfaceTypeData;
   Get the interface type information

function InterfaceUnitName: PShortString;
   Get the unit name of a given interface type information
   - returns '' if this type is not an interface

function IsQWord: boolean;
   Return TRUE if the property is an unsigned 64-bit field

function OrdType: TOldType;
   For ordinal types, get the storage size and sign

function RecordType: PRecordType;
   Get the record type information

function SetEnumType: PEnumType;
   For set types, get the type information of the corresponding enumeration

procedure InterfaceAncestors(out Ancestors: PTypeInfoDynArray; OnlyImplementedBy: TInterfacedObjectClass; out AncestorsImplementedEntry: TPointerDynArray);
   Get all ancestors/parents of a given interface type information
   - only ancestors with an associated TGUID will be added
   - if Only ImplementedBy is not nil, only the interface explicitly implemented by this class will be added, and Ancestors ImplementedEntry[] will contain the corresponding PInterfaceEntry values

TPropInfo = object(TObject)
   A wrapper containing a RTTI property definition
   - used for direct Delphi / UTF-8 SQL type mapping/conversion
   - doesn't depend on RTL's TypInfo unit, to enhance cross-compiler support

Used for DI-2.1.3 (page 2546).
GetProc: PtrUInt;
Contains the offset of a field, or the getter method set by 'read' declaration
- if this field is 0 (no 'read' was specified), raw access methods will use SetProc to get the field memory address to read from
- call TPropInfo.Getter for cross-compiler access to this information

Index: Integer;
Contains the index value of an indexed class data property
- outside SQLite3, this can be used to define a VARCHAR() length value for the textual field definition (sftUTF8Text/sftAnsiText); e.g. the following will create a NAME VARCHAR(40) field:
  Name: RawUTF8 index 40 read fName write fName;
- is used by a dynamic array property for fast usage of the TSQLRecord.DynArray(DynArrayFieldIndex) method

Name: ShortString;
The property definition Name

NameIndex: SmallInt;
Contains the default value (NO_DEFAULT=$80000000 indicates none set) when an ordinal or set property is saved as TPersistent
- see TPropInfo.DefaultOr0/DefaultOrVoid for easy use index of the property in the current inherited class definition
- first name index at a given class level is 0
- index is reset to 0 at every inherited class level

PropType: PTypeInfo;
The type definition of this property
- call TPropInfo.TypeInfo for cross-compiler access to this information

SetProc: PtrUInt;
Contains the offset of a field, or the setter method set by 'write' declaration
- if this field is 0 (no 'write' was specified), raw access methods will use GetProc to get the field memory address to save into
- call TPropInfo.Setter for cross-compiler access to this information

StoredProc: PtrUInt;
Contains the 'stored' boolean value/method (used in TPersistent saving)
- either integer(True) - the default, integer(False), reference to a Boolean field, or reference to a parameterless method that returns a Boolean value
- if a property is marked as "stored AS UNIQUE" (i.e. "stored false"), it is created as UNIQUE in the SQL database and its bit is set in Model.fIsUnique[]
- call TPropInfo.IsStored for cross-compiler access to this information
function GetDoubleProp(Instance: TObject): double;
Raw retrieval of tkFloat/double
- this method will check if the corresponding property is floating-point
- return 0 on any error

function GetDoubleValue(Instance: TObject): double;
Low-level getter of the floating-point property value of a given instance
- this method will check if the corresponding property is floating-point
- return 0 on any error

function GetDynArray(Instance: TObject): TDynArray; overload;
Low-level getter of a dynamic array wrapper
- this method will NOT check if the property is a dynamic array: caller must have already checked that PropType^^.Kind=tkDynArray

function DynamicArrayIsObjArray: boolean;
Return TRUE if this dynamic array has been registered as a T*ObjArray
- the T*ObjArray dynamic array should have been previously registered via TJSONSerializer.RegisterObjArrayForJSON() overloaded methods

function DynamicArrayIsObjArrayInstance: PClassInstance;
Return class instance creation information about a T*ObjArray
- the T*ObjArray dynamic array should have been previously registered via TJSONSerializer.RegisterObjArrayForJSON() overloaded methods
- returns nil if the supplied type is not a registered T*ObjArray
- you can create a new item instance just by calling result^.CreateNew

function GetCurrencyProp(Instance: TObject): currency;
Raw retrieval of tkFloat/currency
- use instead GetCurrencyValue

function GetCurrencyValue(Instance: TObject): Currency;
Low-level getter of the currency property value of a given instance
- this method will check if the corresponding property is exactly currency
- return 0 on any error

function DefaultOr0: integer;
Return the Default RTTI value defined for this property, or 0 if not set

function ClassFromJSON(Instance: TObject; From: PUTF8Char; var Valid: boolean; Options: TJSONToObjectOptions=[]): PUTF8Char;
Read an TObject published property, as saved by ObjectToJSON() function
- will use direct in-memory reference to the object, or call the corresponding setter method (if any), creating a temporary instance via TTypeInfo.ClassCreate
- unserialze the JSON input buffer via a call to JSONToObject()
- by default, a temporary instance will be created if a published field has a setter, and the instance is expected to be released later by the owner class: you can set the j2oSetterExpectsToFreeTempInstance option to let this method release it when the setter returns

function CopyToNewObject(aFrom: TObject): TObject;
Create a new instance of a published property
- copying its properties values from a given instance of another class
- if the destination property is not of the aFrom class, it will first search for any exact match in the destination nested properties
function GetFieldAddr(Instance: TObject): pointer;
  Low-level getter of the field value memory pointer
  - return NIL if both getter and setter are methods

function GetFloatProp(Instance: TObject): double;
  Raw retrieval of tkFloat - with conversion to 64-bit double
  - use instead GetDoubleValue

function GetGenericStringValue(Instance: TObject): string;
  Low-level getter of the long string property value of a given instance
  - uses the generic string type: to be used within the VCL
  - this method will check if the corresponding property is a Long String, or an UnicodeString (for Delphi 2009+), and will return "" if it's not the case

function GetInt64Prop(Instance: TObject): Int64;
  Raw retrieval of tkInt64, tkQWord
  - rather call GetInt64Value

function GetInt64Value(Instance: TObject): Int64;
  Low-level getter of the ordinal property value of a given instance
  - this method will check if the corresponding property is ordinal
  - ordinal properties smaller than tkInt64 will return an Int64-converted value (e.g. tkInteger)
  - return 0 on any error

function GetObjProp(Instance: TObject): TObject;
  Raw retrieval of tkClass

function GetOrdProp(Instance: TObject): PtrInt;
  Raw retrieval of tkInteger, tkEnumeration, tkSet, tkChar, tkWChar, tkBool
  - rather call GetOrdValue/GetInt64Value

function GetOrdValue(Instance: TObject): PtrInt;
  Low-level getter of the ordinal property value of a given instance
  - this method will check if the corresponding property is ordinal
  - return -1 on any error

function Getter(Instance: TObject; Call: PMethod): TPropInfoCall;
  Raw retrieval of the property read access definition
  - note: 'var Call' generated incorrect code on Delphi XE4 -> use PMethod

function GetterAddr(Instance: pointer): pointer;
  Returns the low-level field read address, if GetterIsField is TRUE

function GetterIsField: boolean;
  Return TRUE if the property has no getter but direct field read

function IsBlob: boolean;
  Return true if this property is a BLOB (TSQLRawBlob)

function IsDefaultOrVoid(Instance: TObject): boolean;
  Return TRUE if the property has its Default RTTI value, or is 0/""/nil
  - will call function IsObjectDefaultOrVoid() for class properties
function IsStored(Instance: TObject): boolean;
Return FALSE (AS_UNIQUE) if was marked as "stored AS_UNIQUE" (i.e. "stored false"), or TRUE by default
- if Instance=nil, will work only at RTTI level, not with field or method (and will return TRUE if nothing is defined in the RTTI)

function Next: PPropInfo;
Get the next property information
- no range check: use ClassProp()^.PropCount to determine the properties count
- get the first PPropInfo with ClassProp()^.PropList

function RetrieveFieldSize: integer;
Compute in how many bytes this property is stored

function SameValue(Source: TObject; DestInfo: PPropInfo; Dest: TObject): boolean;
Compare two published properties

function Setter(Instance: TObject; Call: PMethod): TPropInfoCall;
Raw retrieval of the property access definition

function SetterAddr(Instance: pointer): pointer;
Returns the low-level field write address, if SetterIsField is TRUE

function SetterIsField: boolean;
Return TRUE if the property has no setter but direct field write

function TypeInfo: PTypeInfo;
The type information of this property
- will de-reference the PropType pointer on Delphi and newer FPC compilers

function WriteIsDefined: boolean;
Return TRUE if the property has a write setter or direct field

procedure CopyLongStrProp(Source, Dest: TObject);
Raw copy of tkLString

procedure CopyValue(Source, Dest: TObject; DestInfo: PPropInfo=nil);
Copy a published property value from one instance to another
- this method use direct copy of the low-level binary content, and is therefore faster than a SetValue(Dest,GetValue(Source)) call
- if DestInfo is nil, it will assume DestInfo=@self

procedure GetDynArray(Instance: TObject; var result: TDynArray); overload;
Low-level getter of a dynamic array wrapper
- this method will NOT check if the property is a dynamic array: caller must have already checked that PropType^^.Kind=tkDynArray

procedure GetLongStrProp(Instance: TObject; var Value: RawByteString);
Raw retrieval of tkLString
procedure GetLongStrValue(Instance: TObject; var result: RawUTF8);
\[\text{Low-level getter of the long string property value of a given instance}\]
- this method will check if the corresponding property is a Long String, and will return " if it's not the case
- it will convert the property content into RawUTF8, for RawUnicode, WinAnsiString, TSQLRawBlob and generic Delphi 6-2007 string property
- WideString and UnicodeString properties will also be UTF-8 converted

procedure GetRawByteStringValue(Instance: TObject; var Value: RawByteString);
\[\text{Low-level getter of the long string property content of a given instance}\]
- just a wrapper around low-level GetLongStrProp() function
- call GetLongStrValue() method if you want a conversion into RawUTF8
- will work only for Kind=tkLString

procedure GetShortStrProp(Instance: TObject; var Value: RawByteString);
\[\text{Raw retrieval of tkString into an Ansi7String}\]

procedure GetToText(Instance: TObject; WR: TTextWriter; RawUTF8DynArrayAsCSV: boolean=false; Escape: TTextWriterKind=twNone);
\[\text{Low-level appender of the property value to a text buffer}\]
- write the published integer, Int64, floating point values, (wide)string, enumerates (e.g. boolean), variant properties of the object
- dynamic arrays will be serialized as JSON, unless RawUTF8DynArrayAsCSV is set, and a TRawUTF8DynArray property will be stored as CSV

procedure GetVariant(Instance: TObject; var Dest: variant);
\[\text{Low-level getter of the property value into a variant value}\]
- a tkDynArray property is expected to be a T*ObjArray and will be converted into a TDocVariant using a temporary JSON serialization

procedure GetVariantProp(Instance: TObject; var result: Variant);
\[\text{Raw retrieval of tkVariant}\]

procedure GetWideStrProp(Instance: TObject; var Value: WideString);
\[\text{Raw retrieval of tkWString}\]

procedure SetCurrencyProp(Instance: TObject; const Value: Currency);
\[\text{Raw assignment of tkFloat/currency}\]

procedure SetDefaultValue(Instance: TObject; FreeAndNilNestedObjects: boolean=true);
\[\text{Low-level setter of the property value as its default}\]
- this method will check the property type, e.g. setting " for strings, and 0 for numbers, or running FreeAndNil() on any nested object (unless FreeAndNilNestedObjects is false so that ClearObject() is used

procedure SetDoubleProp(Instance: TObject; Value: Double);
\[\text{Raw assignment of tkFloat/double}\]

procedure SetDoubleValue(Instance: TObject; const Value: double);
\[\text{Low-level setter of the floating-point property value of a given instance}\]
- this method will check if the corresponding property is floating-point
**procedure** SetFloatProp(Instance: TObject; Value: TSynExtended);

*Raw assignment of tkFloat*
- use instead SetDoubleValue

**procedure** SetFromText(Instance: TObject; const Text: RawUTF8; TryCustomVariants: PDocVariantOptions=nil; AllowDouble: boolean=false);

*Low-level setter of the property value from its text representation*
- handle published integer, Int64, floating point values, (wide)string, enumerates (e.g. boolean), variant properties of the object
- for variant properties, could unserialize the Text as JSON into a TDocVariantData if TryCustomVariants (and AllowDouble) are set
- dynamic arrays are unserialized from JSON [...], unless a TRawUTF8DynArray property has been stored as CSV

**procedure** SetFromVariant(Instance: TObject; const Value: variant);

*Low-level setter of the property value from a supplied variant*
- will optionally make some conversion if the property type doesn't match the variant type, e.g. a text variant could be converted to integer when setting a tkInteger kind of property
- a tkDynArray property is expected to be a T*ObjArray and will be converted from a TDocVariant using a newly allocated T*ObjArray

**procedure** SetGenericStringValue(Instance: TObject; const Value: string);

*Low-level setter of the string property value of a given instance*
- uses the generic string type: to be used within the VCL
- this method will check if the corresponding property is a Long String or an UnicodeString (for Delphi 2009+), and will call the corresponding SetLongStrValue() or SetUnicodeStrValue() method

**procedure** SetInt64Prop(Instance: TObject; const Value: Int64);

*Raw assignment of tkInt64,tkQWord*
- rather call SetInt64Value

**procedure** SetInt64Value(Instance: TObject; Value: Int64);

*Low-level setter of the ordinal property value of a given instance*
- this method will check if the corresponding property is ordinal

**procedure** SetLongStrProp(Instance: TObject; const Value: RawByteString);

*Raw assignment of tkLString*

**procedure** SetLongStrValue(Instance: TObject; const Value: RawUTF8);

*Low-level setter of the long string property value of a given instance*
- this method will check if the corresponding property is a Long String
- it will convert the property content into RawUTF8, for RawUnicode, WinAnsiString, TSQLRawBlob and generic Delphi 6-2007 string property
- will set WideString and UnicodeString properties from UTF-8 content

**procedure** SetOrdProp(Instance: TObject; Value: PtrInt);

*Raw assignment of tkInteger,tkEnumeration,tkSet,tkChar,tkWChar,tkBool*
- rather call SetOrdValue/SetInt64Value

**procedure** SetOrdValue(Instance: TObject; Value: PtrInt);

*Low-level setter of the ordinal property value of a given instance*
- this method will check if the corresponding property is ordinal
procedure SetVariantProp(Instance: TObject; const Value: Variant);
   Raw assignment of `tkVariant`

procedure SetWideStrProp(Instance: TObject; const Value: WideString);
   Raw assignment of `tkWString`

TReturnInfo = object(TObject)
   A wrapper around method returned result definition
   CallingConvention: TCallingConvention;
      Expected calling convention (only relevant for x86 mode)
   ParamCount: Byte;
      Number of expected parameters
   ParamSize: Word;
      Total size of data needed for stack parameters + 8 (ret-addr + pushed EBP)
   ReturnType: ^PTypeInfo;
      The expected type of the returned function result
      - is nil for procedure
   Version: byte;
      RTTI version
      - 2 up to Delphi 2010, 3 for Delphi XE and up

function Param: PParamInfo;
   Access to the first method parameter definition

TParamInfo = object(TObject)
   A wrapper around an individual method parameter definition
   Flags: TParamFlags;
      The kind of parameter
   Name: ShortString;
      Parameter name
   Offset: Word;
      Parameter offset
      - 0 for EAX, 1 for EDX, 2 for ECX
      - any value >= 8 for stack-based parameter
   ParamType: PTypeInfo;
      The parameter type information

function Next: PParamInfo;
   Get the next parameter information
   - no range check: use TReturnInfo.ParamCount to determine the appropriate count
TMethodInfo = object(TObject)
  A wrapper around a method definition

  Addr: Pointer;
  The associated method code address

  Len: Word;
  Size (in bytes) of this TMethodInfo block

  Name: ShortString;
  Method name

  function MethodAddr: Pointer;
  Wrapper returning nil and avoiding a GPF if @self=nil

  function ReturnInfo: PReturnInfo;
  Retrieve the associated parameters information

PropWrap = packed record
  Used to map a TPropInfo.GetProc/SetProc and retrieve its kind
  - defined here for proper Delphi inlining

  Kind: byte;
  =$ff for a ptField address, or =$fe for a ptVirtual method

TSQLPropInfo = class(TObject)
  Abstract parent class to store information about a published property
  - property information could be retrieved from RTTI (TSQLPropInfoRTTI*), or be defined by code
  (TSQLPropInfoCustom derived classes) when RTTI is not available

  constructor Create(const aName: RawUTF8; aSQLFieldType: TSQLFieldType; aAttributes: TSQLPropInfoAttributes; aFieldWidth, aPropertyIndex: integer); reintroduce;
  virtual;

  Initialize the internal fields
  - should not be called directly, but with dedicated class methods like class function
  TSQLPropInfoRTTI.CreateFrom() or overridden constructors

  function CompareValue(Item1,Item2: TObject; CaseInsensitive: boolean): PtrInt;
  virtual;

  Compare the content of the property of two objects
  - not all kind of properties are handled: only main types (like GetHash)
  - if CaseInsensitive is TRUE, will apply NormToUpper[] 8 bits uppercase, handling RawUTF8
  properties just like the SYSTEMNOCASE collation
  - this method should match the case-sensitivity of GetHash()
  - this default implementation will call GetValueVar() for slow comparison

  function GetFieldAddr(Instance: TObject): pointer; virtual; abstract;

  Returns an untyped pointer to the field property memory in a given instance
function GetHash(Instance: TObject; CaseInsensitive: boolean): cardinal; virtual;
Retrieves an unsigned 32-bit hash of the corresponding property
- Not all kind of properties are handled: only main types
- If CaseInsensitive is TRUE, will apply NormToUpper[] 8 bits uppercase, handling RawUTF8 properties just like the SYSTEMNOCASE collation
- Note that this method can return a hash value of 0
- This method should match the case-sensitivity of CompareValue()
- This default implementation will call GetValueVar() for slow computation

function GetValue(Instance: TObject; ToSQL: boolean; wasSQLString: PBoolean=nil): RawUTF8;
Convert the property value into an UTF-8 encoded text
- If ToSQL is true, result is on SQL form (false->'0' e.g.)
- If ToSQL is false, result is on JSON form (false->'false' e.g.)
- BLOB field returns SQLite3 BLOB literals ("x'01234'" e.g.) if ToSQL is true, or base-64 encoded stream for JSON ("\uFF0base64encodedbinary")
- Getter method (read Get*) is called if available
- Handle Delphi values into UTF-8 SQL conversion
- SftBlobDynArray, sftBlobCustom or sftBlobRecord are returned as BLOB literals ("X'53514C697465'") if ToSQL is true, or base-64 encoded stream for JSON ("\uFF0base64encodedbinary")
- Handle TPersistent, TCollection, TRawUTF8List or TStrings with ObjectToJSON

function IsValueVoid(Instance: TObject): boolean;
Returns TRUE if value is 0 or "

function SetBinary(Instance: TObject; P, PEnd: PAnsiChar): PAnsiChar; virtual; abstract;
Read the property value from a binary buffer
- PEnd should point to the end of the P input buffer, to avoid any overflow
- Returns next char in input buffer on success, or nil in case of invalid content supplied e.g.

function SetFieldSQLVar(Instance: TObject; const aValue: TSQLVar): boolean; virtual;
Set a field value from a TSQLVar value

function SQLDBFieldTypeName: PShortString;
The corresponding column type name, as managed for abstract database access

procedure CopyProp(Source: TObject; DestInfo: TSQLPropInfo; Dest: TObject);
Copy a value from one instance to another property instance
- If the property has been flattened (for a TSQLPropInfoRTTI), the real Source/Dest instance will be used for the copy

procedure CopyValue(Source, Dest: TObject); virtual;
Copy a property value from one instance to another
- Both objects should have the same exact property

procedure GetBinary(Instance: TObject; W: TFileBufferWriter); virtual; abstract;
Append the property value into a binary buffer
procedure GetFieldSQLVar(Instance: TObject; var aValue: TSQLVar; var temp: RawByteString); virtual;

Retrieve a field value into a TSQLVar value
- the temp RawByteString is used as a temporary storage for TEXT or BLOB and should be available during all access to the TSQLVar fields

procedure GetJSONValues(Instance: TObject; W: TJSONSerializer); virtual;

Add the JSON content corresponding to the given property
- this default implementation will call safe but slow GetValueVar() method

procedure GetValueVar(Instance: TObject; ToSQL: boolean; var result: RawUTF8; wasSQLString: PBoolean); virtual; abstract;

Convert the property value into an UTF-8 encoded text
- this method is the same as GetValue(), but avoid assigning the result string variable (some speed up on multi-core CPUs, since avoid a CPU LOCK)
- this virtual method is the one to be overridden by the implementing classes

procedure GetVariant(Instance: TObject; var Dest: Variant); virtual; abstract;

Retrieve the property value into a Variant
- will set the Variant type to the best matching kind according to the SQLFieldType type
- BLOB field returns SQLite3 BLOB textual literals ("x'01234'" e.g.)
- dynamic array field is returned as a variant array

procedure NormalizeValue(var Value: RawUTF8); virtual; abstract;

Normalize the content of Value, so that GetValue(Object,true) should return the same content (true for ToSQL format)

procedure SetValue(Instance: TObject; Value: PUTF8Char; wasString: boolean); virtual; abstract;

Convert UTF-8 encoded text into the property value
- setter method (write Set*) is called if available
- if no setter exists (no write declaration), the gotted field address is used
- handle UTF-8 SQL to Delphi values conversion
- expect BLOB fields encoded as SQLite3 BLOB textual literals ("x'01234'" e.g.) or base-64 encoded stream for JSON ("\uFFF0base64encodedbinary") - i.e. both format supported by BlobToTSQLRawBlob() function
- handle TPersistent, TCollection, TRawUTF8List or TStrings with JSONToObject
- note that the supplied Value buffer won't be modified by this method: overridden implementation should create their own temporary copy

procedure SetValueVar(Instance: TObject; const Value: RawUTF8; wasString: boolean); virtual;

Convert UTF-8 encoded text into the property value
- just a wrapper around SetValue(...,pointer(Value),...) which may be optimized for overridden methods

procedure SetVariant(Instance: TObject; const Source: Variant); virtual;

Set the property value from a Variant value
- dynamic array field must be set from a variant array
- will set the Variant type to the best matching kind according to the SQLFieldType type
- expect BLOB fields encoded as SQLite3 BLOB literals ("x'01234'" e.g.)
property Attributes: TSQLPropInfoAttributes read fAttributes write fAttributes;
The ORM attributes of this property
- contains as lUnique e.g for TSQLRecord published properties marked as
  property MyProperty: RawUTF8 stored AS_UNIQUE;
  (i.e. "stored false")

property FieldWidth: integer read fFieldWidth;
The optional width of this field, in external databases
- is set e.g. by index attribute of TSQLRecord published properties as
  property MyProperty: RawUTF8 index 18;

property Name: RawUTF8 read fName;
The property definition Name

property NameDisplay: string read GetNameDisplay;
The property definition Name, after un-camelcase and translation

property NameUnflattened: RawUTF8 read fNameUnflattened;
The property definition Name, with full path name if has been flattened
- if the property has been flattened (for a TSQLPropInfoRTTI), the real full nested class will be
  returned, e.g. 'Address.Country.Is0' for the 'Address_Country' flattened property name

property PropertyIndex: integer read fPropertyIndex;
The property index in the RTTI

property SQLDBFieldType: TSQLDBFieldType read fSQLDBFieldType;
The corresponding column type, as managed for abstract database access
- TNullable* fields will report here the corresponding simple DB type, e.g. ftInt64 for
  TNullableInteger (following SQLFieldTypeStored value)

property SQLFieldRTTITypeName: RawUTF8 read GetSQLFieldRTTITypeName;
The type name, as defined in the RTTI
- returns e.g. 'RawUTF8'
- will return the TSQLPropInfo class name if it is not a TSQLPropInfoRTTI

property SQLFieldType: TSQLFieldType read fSQLFieldType;
The corresponding column type, as managed by the ORM layer

property SQLFieldTypeName: PShortString read GetSQLFieldTypeName;
The corresponding column type name, as managed by the ORM layer and retrieved by the RTTI
- returns e.g. 'sftTimeLog'

property SQLFieldTypeStored: TSQLFieldType read fSQLFieldTypeStored;
The corresponding column type, as stored by the ORM layer
- match SQLFieldType, unless for SQLFieldType=sftNullable, in which this field will contain the
  simple type eventually stored in the database

TSQLPropInfoRTTI = class(TSQLPropInfo)
Parent information about a published property retrieved from RTTI
constructor Create(aPropInfo: PPropInfo; aPropIndex: integer; aSQLFieldType: TSQLFieldType; aOptions: TSQLPropInfoListOptions); reintroduce; virtual;

  Initialize the internal fields
  - should not be called directly, but with dedicated class methods like class function CreateFrom()

class function CreateFrom(aPropInfo: PPropInfo; aPropIndex: integer; aOptions: TSQLPropInfoListOptions; const aFlattenedProps: PPropInfoDynArray): TSQLPropInfo;

  This meta-constructor will create an instance of the exact descendant of the specified property RTTI
  - it will raise an EORMException in case of an unhandled type

function Flattened(Instance: TObject): TObject;

  For pilSubClassesFlattening properties, compute the actual instance containing the property value
  - if the property was not flattened, return the instance

function GetFieldAddr(Instance: TObject): pointer; override;

  Generic way of implementing it

procedure GetVariant(Instance: TObject; var Dest: Variant); override;

  Retrieve the property value into a Variant
  - will set the Variant type to the best matching kind according to the SQLFieldType type
  - BLOB field returns SQLite3 BLOB textual literals ("x'01234'' e.g.)
  - dynamic array field is returned as a variant array

class procedure RegisterTypeInfo(aTypeInfo: Pointer);

  Register this class corresponding to the RTTI TypeInfo() pointer
  - could be used e.g. to define custom serialization and process of any custom type

property FlattenedPropInfo: PPropInfoDynArray read fFlattenedProps;

  For pilSubClassesFlattening properties, the parents RTTI

property PropInfo: PPropInfo read fPropInfo;

  Corresponding RTTI information

property PropType: PTypeInfo read fPropType;

  Corresponding type information, as retrieved from PropInfo RTTI

TSQLPropInfoRTTIInt32 = class(TSQLPropInfoRTTI)
  Information about an ordinal Int32 published property

TSQLPropInfoRTTISet = class(TSQLPropInfoRTTIInt32)
  Information about a set published property

TSQLPropInfoRTTIEnum = class(TSQLPropInfoRTTIInt32)
  Information about a enumeration published property
  - can be either sftBoolean or sftEnumerate kind of property

TSQLPropInfoRTTIChar = class(TSQLPropInfoRTTIInt32)
  Information about a character published property

TSQLPropInfoRTTIInt64 = class(TSQLPropInfoRTTI)
  Information about an ordinal Int64 published property
TSQLPropInfoRTTITimeLog = class(TSQLPropInfoRTTIInt64)
  Information about a TTimeLog published property
  - stored as an Int64, but with a specific class

TSQLPropInfoRTTIUnixTime = class(TSQLPropInfoRTTIInt64)
  Information about a TUnixTime published property
  - stored as an Int64, but with a specific class

TSQLPropInfoRTTIUnixMSTime = class(TSQLPropInfoRTTIInt64)
  Information about a TUnixMSTime published property
  - stored as an Int64, but with a specific class

TSQLPropInfoRTTIDouble = class(TSQLPropInfoRTTI)
  Information about a floating-point Double published property

TSQLPropInfoRTTICurrency = class(TSQLPropInfoRTTIDouble)
  Information about a fixed-decimal Currency published property

TSQLPropInfoRTTIDateTime = class(TSQLPropInfoRTTIDouble)
  Information about a TDateTime published property

TSQLPropInfoRTTIAnsi = class(TSQLPropInfoRTTI)
  Information about a AnsiString published property

TSQLPropInfoRTTIRawUTF8 = class(TSQLPropInfoRTTIAnsi)
  Information about a RawUTF8 published property
  - will also serialize a RawJSON property without JSON escape

TSQLPropInfoRTTIRawUnicode = class(TSQLPropInfoRTTIAnsi)
  Information about a RawUnicode published property

TSQLPropInfoRTTIRawBlob = class(TSQLPropInfoRTTIAnsi)
  Information about a TSQLRawBlob published property

TSQLPropInfoRTTIWide = class(TSQLPropInfoRTTI)
  Information about a WideString published property

TSQLPropInfoRTTIDynArray = class(TSQLPropInfoRTTI)
  Information about a dynamic array published property

  constructor Create(aPropInfo: PPropInfo; aPropIndex: integer; aSQLFieldType: TSQLFieldType; aOptions: TSQLPropInfoListOptions); override;
    Initialize the internal fields
    - should not be called directly, but with dedicated class methods like class function CreateFrom()

  property DynArrayElemType: PTypeInfo read GetDynArrayElemType;
    Read-only access to the low-level type information the array item type

  property DynArrayIndex: integer read fFieldWidth;
    Optional index of the dynamic array published property
    - used e.g. for fast lookup by TSQLRecord.DynArray(DynArrayFieldIndex)
property ObjArray: PClassInstance read fObjArray;

Dynamic array item information for a T*ObjArray
- equals nil if this dynamic array was not previously registered via TJSONSerializer.RegisterObjArrayForJSON()
- note that if the field is a T*ObjArray, you could create a new item by calling ObjArray^.CreateNew
- T*ObjArray database column will be stored as text

TSQLOptionInfoRTTIVariant = class(TSQLOptionInfoRTTI)
Information about a variant published property
- is also used for TNullable* properties

constructor Create(aPropInfo: PPropInfo; aPropIndex: integer; aSQLFieldType: TSQLFieldType; aOptions: TSQLPropInfoListOptions); override;

Initialize the internal fields

property DocVariantOptions: TDocVariantOptions read fDocVariantOptions write fDocVariantOptions;

How this property will deal with its instances (including TDocVariant)
- by default, contains JSON_OPTIONS_FAST for best performance - i.e. [dvoReturnNullForUnknownProperty,dvoValueCopiedByReference]
- set JSON_OPTIONS_FAST_EXTENDED (or include dvoSerializeAsExtendedJson) so that any TDocVariant nested field names will not be double-quoted, saving some chars in the stored TEXT column and in the JSON escaped transmitted data over REST, by writing '{"name":"John","age":123}' instead of ["name":"John","age":123]': be aware that this syntax is supported by the ORM, SOA, TDocVariant, TBSONVariant, and our SynCrossPlatformJSON unit, but not AJAX/JavaScript or most JSON libraries
- see also TSQLModel/TSQLRecordProperties.SetVariantFieldsDocVariantOptions

TSQLOptionInfoCustom = class(TSQLOptionInfo)
Abstract information about a record-like property defined directly in code
- do not use this class, but TSQLPropInfoRecordRTTI and TSQLPropInfoRecordFixedSize
- will store the content as BLOB by default, and SQLFieldType as sftBlobCustom
- if aData2Text/aText2Data are defined, use TEXT storage and sftUTF8Custom type

constructor Create(const aName: RawUTF8; aSQLFieldType: TSQLFieldType; aAttributes: TSQLPropInfoAttributes; aFieldWidth, aPropIndex: Integer; aProperty: pointer; aData2Text: TOnSQLPropInfoRecord2Text; aText2Data: TOnSQLPropInfoRecord2Data); reintroduce;

Define a custom property in code
- do not call this constructor directly, but one of its inherited classes, via a call to TSQLRecordProperties.RegisterCustom*()

TSQLPropInfoRecordTyped = class(TSQLOptionInfoCustom)
Information about a record property defined directly in code using RTTI
TSQLPropInfoRecordRTTI = class(TSQLPropInfoRecordTyped)

Information about a record property defined directly in code
- Delphi does not publish RTTI for published record properties
- you can use this class to register a record property from its RTTI
- will store the content as BLOB by default, and SQLFieldType as sftBlobCustom
- if aData2Text/aText2Data are defined, use TEXT storage and sftUTF8Custom type
- this class will use only binary RecordLoad/RecordSave methods

constructor Create(aRecordInfo: PTypeInfo; const aName: RawUTF8; aPropertyIndex: integer; aPropertyPointer: pointer; aAttributes: TSQLPropInfoAttributes=[]; aFieldWidth: integer=0; aData2Text: TOnSQLPropInfoRecord2Text=nil; aText2Data: TOnSQLPropInfoRecord2Data=nil); reintroduce; overload;

Define a record property from its RTTI definition
- handle any kind of record with available generated TypeInfo()
- aPropertyPointer shall be filled with the offset to the private field within a nil object, e.g for
  class TMainObject = class(TSQLRecord)
      (...)
      ffieldName: TMyRecord;
      public
          (...)
          property FieldName: TMyRecord read ffieldName write ffieldName;
  end;
you will have to register it via a call to
TSQLRecordProperties.RegisterCustomRTTIRecordProperty()
- optional aIsNotUnique parameter can be defined
- implementation will use internally RecordLoad/RecordSave functions
- you can specify optional aData2Text/aText2Data callbacks to store the content as textual values, and not as BLOB

TSQLPropInfoRecordFixedSize = class(TSQLPropInfoRecordTyped)

Information about a fixed-size record property defined directly in code
- Delphi does not publish RTTI for published record properties
- you can use this class to register a record property with no RTTI (i.e. a record with no reference-counted types within)
- will store the content as BLOB by default, and SQLFieldType as sftBlobCustom
- if aData2Text/aText2Data are defined, use TEXT storage and sftUTF8Custom type

constructor Create(aRecordSize: cardinal; const aName: RawUTF8; aPropertyIndex: integer; aPropertyPointer: pointer; aAttributes: TSQLPropInfoAttributes=[]; aFieldWidth: integer=0; aData2Text: TOnSQLPropInfoRecord2Text=nil; aText2Data: TOnSQLPropInfoRecord2Data=nil); reintroduce; overload;

Define an unmanaged fixed-size record property
- simple kind of records (i.e. those not containing reference-counted members) do not have RTTI generated, at least in older versions of Delphi: use this constructor to define a direct property access
- main parameter is the record size, in bytes
**TSQLPropInfoCustomJSON** = `class` (TSQLPropInfoRecordTyped)

*Information about a custom property defined directly in code*
- you can define any kind of property, either a record or any type
- this class will use JSON serialization, by type name or TypeInfo() pointer
- will store the content as TEXT by default, and SQLFieldType as sftUTF8Custom

**constructor** Create(`const` aTypeName, aName: RawUTF8; aPropertyIndex: integer; aPropertyPointer: pointer; aAttributes: TSQLPropInfoAttributes=[]; aFieldWidth: integer=0); `reintroduce`; overload;

*Define a custom property from its RTTI definition*
- handle any kind of property, e.g. from enhanced RTTI or a custom record defined via TTextWriter.RegisterCustomJSONSerializer[FromText]()
- aPropertyPointer shall be filled with the offset to the private field within a nil object, e.g for

```pascal
class TMainObject = class(TSQLRecord)
  (...)
public
  (...)
  property GUID: TGUID read fGUID write fGUID;
end;
```

you will have to register it via a call to
TSQLRecordProperties.RegisterCustomPropertyFromTypeName()
- optional aIsNotUnique parameter can be defined
- implementation will use internally RecordLoadJSON/RecordSave functions
- you can specify optional aData2Text/aData2Text2Data callbacks to store the content as textual values, and not as BLOB

**constructor** Create(aTypeInfo: PTypeInfo; `const` aName: RawUTF8; aPropertyIndex: integer; aPropertyPointer: pointer; aAttributes: TSQLPropInfoAttributes=[]; aFieldWidth: integer=0); `reintroduce`; overload;

*Define a custom property from its RTTI definition*
- handle any kind of property, e.g. from enhanced RTTI or a custom record defined via TTextWriter.RegisterCustomJSONSerializer[FromText]()
- aPropertyPointer shall be filled with the offset to the private field within a nil object, e.g for

```pascal
class TMainObject = class(TSQLRecord)
  (...)
public
  (...)
  property FieldName: TMyRecord read fFieldName write fFieldName;
end;
```

you will have to register it via a call to
TSQLRecordProperties.RegisterCustomPropertyFromRTTI()
- optional aIsNotUnique parameter can be defined
- implementation will use internally RecordLoadJSON/RecordSave functions
- you can specify optional aData2Text/aData2Text2Data callbacks to store the content as textual values, and not as BLOB

**constructor** Create(aPropInfo: PPropInfo; aPropIndex: integer); `reintroduce`; overload; `virtual`;

*Initialize the internal fields*
- should not be called directly
destructor Destroy; override;

Finalize the instance

property CustomParser: TJSONCustomParserRTTI read fCustomParser;

The corresponding custom JSON parser

TSQLPropInfoList = class(TObject)

Handle a read-only list of fields information for published properties
- is mainly used by our ORM for TSQLRecord RTTI, but may be used for any TPersistent

constructor Create(aTable: TClass; aOptions: TSQLPropInfoListOptions);
Initialize the list from a given class RTTI

destructor Destroy; override;

Release internal list items

function Add(aItem: TSQLPropInfo): integer;
Add a TSQLPropInfo to the list

functionByName(aName: PUTF8Char): TSQLPropInfo; overload;
Find an item in the list
- returns nil if not found

function ByRawUTF8Name(const aName: RawUTF8): TSQLPropInfo; overload;
Find an item in the list
- returns nil if not found

function IndexByName(aName: PUTF8Char): integer; overload;
Find an item in the list
- returns -1 if not found

function IndexByName(const aName: RawUTF8): integer; overload;
Find an item in the list
- returns -1 if not found

function IndexByNameOrExcept(const aName: RawUTF8): integer;
Find an item by name in the list, including RowID/ID
- will identify 'ID' / 'RowID' field name as -1
- raise an EORMException if not found in the internal list

function IndexByNameUnflattenedOrExcept(const aName: RawUTF8): integer;
Find an item in the list, searching by unflattened name
- for a flattened property, you may for instance call
  IndexByNameUnflattenedOrExcept('Address.Country.Iso') instead of
  IndexByNameOrExcept('Address_Country')
- won't identify 'ID' / 'RowID' field names, just List[].
- raise an EORMException if not found in the internal list

procedure IndexesByNamesOrExcept(const aNames: array of RawUTF8; const aIndexes: array of PInteger);
Find one or several items by name in the list, including RowID/ID
- will identify 'ID' / 'RowID' field name as -1
- raise an EORMException if not found in the internal list
procedure NamesToRawUTF8DynArray(var Names: TRawUTF8DynArray);

*Fill a TRawUTF8DynArray instance from the field names*
- excluding ID

**property** Count: integer **read** fCount;

*Returns the number of TSQLPropInfo in the list*

**property** Items[aIndex: integer]: TSQLPropInfo **read** GetItem;

*Read-only retrieval of a TSQLPropInfo item*
- will raise an exception if out of range

**property** List: TSQLPropInfoObjArray **read** fList;

*Quick access to the TSQLPropInfo list*
- note that length(List) may not equal Count, since is its capacity

TINIWriter = **class**(TTextWriter)

*Simple writer to a Stream, specialized for writing an object as INI*
- resulting content will be UTF-8 encoded
- use an internal buffer, faster than string+string

procedure WriteObject(Value: TObject; **const** SubCompName: RawUTF8=''; WithSection: boolean=true; RawUTF8DynArrayAsCSV: boolean=false; **reintroduce**);

*Write the published properties of the object in INI text format*
- i.e. append PropertyName=PropertyValue lines
- add a new INI-like section with [Value.ClassName] if WithSection is true
- use internally TPropInfo.GetToText for the conversion to text
- Value object must have been compiled with the $M+ define, i.e. must inherit from TPersistent, TSynPersistent or TSQLRecord
- the enumerates properties are stored with their integer index value
- dynamic arrays will be serialized as JSON, unless RawUTF8DynArrayAsCSV is set, and a TRawUTF8DynArray property will be stored as CSV
- content can be read back using overloaded procedures ReadObject()

TJSONSerializer = **class**(TJSONWriter)

*Simple writer to a Stream, specialized for writing an object as JSON*
- override WriteObject() to use class RTTI process of this unit, and allow custom JSON serialization
- this is the full-feature JSON serialization class

**class function** RegisterObjArrayFindType(aDynArray: PTypeInfo): PClassInstance;

*Retrieve TClassInstance information for a T*ObjArray dynamic array type*
- the T*ObjArray dynamic array should have been previously registered via TJSONSerializer.RegisterObjArrayForJSON() overloaded methods
- returns nil if the supplied type is not a registered T*ObjArray

**class function** RegisterObjArrayFindTypeInfo(aClass: TClass): PTypeInfo;

*Retrieve the T*ObjArray dynamic array type RTTI for a given item class*
- the T*ObjArray dynamic array should have been previously registered via TJSONSerializer.RegisterObjArrayForJSON() overloaded methods
- returns nil if the supplied type is not a registered T*ObjArray
class procedure RegisterClassForJSON(const aItemClass: array of TClass); overload;

Let a given class be recognized by JSONToObject() from "ClassName":".."
- TObjectList item instances will be created corresponding to the serialized class name field specified, and JSONToObject() can create a new instance using the "ClassName":"..." field to identify the class type
- by default, all referenced TSQLRecord classes will be globally registered when TSQLRecordProperties information is retrieved
- this method is thread-safe, but should be called before any serialization

class procedure RegisterClassForJSON(aItemClass: TClass); overload;

Let a given class be recognized by JSONToObject() from "ClassName":"..
- TObjectList item instances will be created corresponding to the serialized class name field specified, and JSONToObject() can create a new instance using the "ClassName":"..." field to identify the class type
- by default, all referenced TSQLRecord classes will be globally registered when TSQLRecordProperties information is retrieved
- this method is thread-safe, but should be called before any serialization

class procedure RegisterCollectionForJSON(aCollection: TCollectionClass; aItem: TCollectionItemClass);

Let a given TCollection be recognized during JSON serialization
- due to how TCollection instances are created, you can not create a server-side instance of TCollection directly
- first workaround is to inherit from TInterfacedCollection
- this method allows to recognize the needed TCollectionItem class for a given TCollection class, so allow to (un)serialize any TCollection, without defining a new method and inherits from TInterfacedCollection
- note that both supplied classes will be registered for the internal "ClassName":"..."
- RegisterClassForJSON() process
- this method is thread-safe, but should be called before any serialization

class procedure RegisterCustomSerializer(aClass: TClass; aReader: TJSONSerializerCustomReader; aWriter: TJSONSerializerCustomWriter);

Define a custom serialization for a given class
- by default, TSQLRecord, TPersistent, TStrings, TCollection classes are processed: but you can specify here some callbacks to perform the serialization process for any class
- any previous registration is overridden
- setting both aReader=aWriter=nil will return back to the default class serialization (i.e. published properties serialization)
- note that any inherited classes will be serialized as the parent class
- this method is thread-safe, but should be called before any serialization
**class procedure** RegisterCustomSerializerFieldNames(aClass: TClass; const aClassFields, aJsonFields: array of ShortString);

*Define custom serialization of field names for a given class*
- any aClassField[] property name will be serialized using aJsonFields[]
- if any aJsonFields[] equals "", this published property will be excluded from the serialization object
- aJsonFields[] is expected to be only plain pascal identifier, i.e. A-Z a-z 0-9 and _ characters, up to 63 in length
- setting both aClassField=aJsonFields=[] will return back to the default class serialization (i.e. serialization with published properties names)
- by design, this customization excludes RegisterCustomSerializer() with custom reader/writer callbacks
- note that any inherited classes will be serialized as the parent class
- this method is thread-safe, but should be called before any serialization

**class procedure** RegisterObjArrayForJSON(const aDynArrayClassPairs: array of const);

*Let T*ObjArray dynamic arrays be used for storage of class instances*
- will allow JSON serialization and unserialization of the registered dynamic array property defined in any TPersistent or TSQLRecord
- will call the overloaded RegisterObjArrayForJSON() class method by pair:
  TJSONSerializer.RegisterObjArrayForJSON([TypeInfo(TAddressObjArray),TAddress, TypeInfo(TUserObjArray),TUser]);

**class procedure** RegisterObjArrayForJSON(aDynArray: PTypeInfo; aItem: TClass; aReader: TDynArrayJSONCustomReader=nil; aWriter: TDynArrayJSONCustomWriter=nil);

*Let a T*ObjArray dynamic array be used for storage of class instances*
- will allow JSON serialization and unserialization of the registered dynamic array property defined in any TPersistent or TSQLRecord
- could be used as such (note the T*ObjArray type naming convention):
  TUserObjArray = array of TUser;
  ...
  TJSONSerializer.RegisterObjArrayForJSON(TypeInfo(TUserObjArray),TUser);
- then you can use ObjArrayAdd/ObjArrayFind/ObjArrayDelete to manage the stored items, and never forget to call ObjArrayClear to release the memory
- will use the default published properties serializer, unless you specify your custom Reader/Write callbacks
procedure WriteObject(Value: TObject; Options: TTextWriterWriteObjectOptions=[woDontStoreDefault]); override;

Serialize as JSON the published integer, Int64, floating point values, TDateTime (stored as ISO 8601 text), string and enumerate (e.g. boolean) properties of the object
- won't handle shortstring properties
- the object must have been compiled with the $M+ define, i.e. must inherit from TPersistent or TSQLRecord, or has been defined with a custom serializer via RegisterCustomSerializer()
- will write also the properties published in the parent classes
- the enumerates properties are stored with their integer index value by default, but will be written as text if woFullExpand option is set
- TList objects are not handled by default - they will be written only if FullExpand is set to true (and JSONToObject won't be able to read it)
- nested properties are serialized as nested JSON objects
- any TCollection property will also be serialized as JSON array
- any TString or TRawUTF8List property will also be serialized as JSON string array
- function ObjectToJSON() is just a wrapper over this method

property SQLRecordOptions: TJSONSerializerSQLRecordOptions read fSQLRecordOptions write SetSQLRecordOptions;

Customize TSQLRecord.GetJSONValues serialization process
- jwoAsJsonNotAsString will force TSQLRecord.GetJSONValues to serialize nested property instances as a JSON object/array, not a JSON string: i.e. root/table/id REST will be ready-to-be-consumed from AJAX clients (e.g. TSQLPropInfoRTTIOBJECT.GetJSONValues as a JSON object, and TSQLPropInfoRTTIArray.GetJSONValues as a JSON array)
- jwoldID_str will add an "ID_str":"12345" property to the default "ID":12345 field to circumvent JavaScript's limitation of 53-bit for integer numbers, which is easily reached with our 64-bit TID values, e.g. if TSynUniqueIdentifier are used to generate the IDs: AJAX clients should better use this "ID_str" string value to identify each record, and ignore the "id" fields

TInterfacedCollection = class(TCollection)

Any TCollection used between client and server shall inherit from this class
- you should override the GetClass virtual method to provide the expected collection item class to be used on server side
- another possibility is to register a TCollection/TCollectionItem pair via a call to TJSONSerializer.RegisterCollectionForJSON()

constructor Create; reintroduce; virtual;

This constructor which will call GetClass to initialize the collection
TCollectionItemAutoCreateFields = class(TCollectionItem)
Abstract TCollectionItem class, which will instantiate all its nested class published properties, then release them (and any T*ObjArray) when freed
- could be used for gathering of TCollectionItem properties, e.g. for Domain objects in DDD, especially for list of value objects
- consider using T*ObjArray dynamic array published properties in your value types instead of TCollection storage: T*ObjArray have a lower overhead and are easier to work with, once TJSONSerializer.RegisterObjArrayForJSON is called to register the T*ObjArray type
- note that non published (e.g. public) properties won’t be instantiated, serialized, nor released - but may contain weak references to other classes
- please take care that you will not create any endless recursion: you should ensure that at one level, nested published properties won’t have any class instance refering to its owner (there is no weak reference - remember!)
- since the destructor will release all nested properties, you should never store a reference to any of those nested instances if this owner may be freed before

constructor Create(Collection: TCollection); override;
This overriden constructor will instantiate all its nested TPersistent/TSynPersistent/TSynAutoCreateFields published properties

destructor Destroy; override;
Finalize the instance, and release its published properties

TPersistentAutoCreateFields = class(TPersistentWithCustomCreate)
Abstract TPersistent class, which will instantiate all its nested TPersistent class published properties, then release them (and any T*ObjArray) when freed
- TSynAutoCreateFields is to be preferred in most cases, thanks to its lower overhead
- note that non published (e.g. public) properties won’t be instantiated, serialized, nor released - but may contain weak references to other classes
- please take care that you will not create any endless recursion: you should ensure that at one level, nested published properties won’t have any class instance refering to its owner (there is no weak reference - remember!)
- since the destructor will release all nested properties, you should never store a reference to any of those nested instances if this owner may be freed before

constructor Create; override;
This overriden constructor will instantiate all its nested TPersistent/TSynPersistent/TSynAutoCreateFields published properties

destructor Destroy; override;
Finalize the instance, and release its published properties
**TSynAutoCreateFields = class (TSynPersistent)**

*Our own empowered TPersistentAutoCreateFields-like parent class*
- this class is a perfect parent to store any data by value, e.g. DDD Value Objects, Entities or Aggregates
- is defined as an abstract class able with a virtual constructor, RTTI for published properties, and automatic memory management of all nested class published properties: any class defined as a published property will be owned by this instance - i.e. with strong reference
- will also release any T*ObjArray dynamic array storage of persistents, previously registered via TJSONSerializer.RegisterObjArrayForJSON()
- nested published classes (or T*ObjArray) don't need to inherit from TSynAutoCreateFields: they may be from any TPersistent/TPersistentAutoCreateFields type
- note that non published (e.g. public) properties won't be instantiated, serialized, nor released - but may contain weak references to other classes
- please take care that you will not create any endless recursion: you should ensure that at one level, nested published properties won't have any class instance refering to its owner (there is no weak reference - remember!)
- since the destructor will release all nested properties, you should never store a reference to any of those nested instances if this owner may be freed before
- TPersistent/TPersistentAutoCreateFields have an unexpected speed overhead due a giant lock introduced to manage property name fixup resolution (which we won't use outside the VCL) - this class is definitively faster

**destructor Destroy; override;**

*Finalize the instance, and release its published properties*

**class function NewInstance: TObject; override;**

*This overridden constructor will instantiate all its nested TPersistent/TSynPersistent/TSynAutoCreateFields published properties*

**procedure AfterLoad; virtual;**

*Virtual method allowing instance customization after initialization*
- called e.g. by JsonToObject, but may be executed after manual fields assignment
- do nothing by default
- may be overridden for string interning or content customization

**TSynAutoCreateFieldsLocked = class (TSynAutoCreateFields)**

*Adding locking methods to a TSynAutoCreateFields with virtual constructor*

**constructor Create; override;**

*Initialize the object instance, and its associated lock*

**destructor Destroy; override;**

*Release the instance (including the locking resource)*

**procedure Lock;**

*Could be used as a short-cut to Safe.Lock*

**procedure UnLock;**

*Could be used as a short-cut to Safe.UnLock*
property Safe: TSynLocker read fSafe;

Access to the locking methods of this instance
- use Safe.Lock/TryLock with a try ... finally Safe.Unlock block

TInterfacedObjectAutoCreateFields =
class(TInterfacedObjectWithCustomCreate)
Abstract TInterfacedObject class, which will instantiate all its nested TPersistent/TSynPersistent published properties, then release them when freed
- will handle automatic memory management of all nested class and T*ObjArray published properties: any class or T*ObjArray defined as a published property will be owned by this instance
- i.e. with strong reference
- non published properties (e.g. public) won't be instantiated, so may store weak class references
- could be used for gathering of TCollectionItem properties, e.g. for Domain objects in DDD, especially for list of value objects, with some additional methods defined by an Interface
- since the destructor will release all nested properties, you should never store a reference to any of those nested instances if this owner may be freed before

constructor Create; override;
This overridden constructor will instantiate all its nested
TPersistent/TSynPersistent/TSynAutoCreateFields class and T*ObjArray published properties

destructor Destroy; override;
Finalize the instance, and release its published properties

TSynJsonFileSettings = class(TSynAutoCreateFields)
Abstract parent class able to store settings as JSON file

function LoadFromFile(const aFileName: TFileName): boolean; virtual;
Read existing settings from a JSON file

function LoadFromJson(var aJson: RawUTF8): boolean;
Read existing settings from a JSON content

procedure SaveIfNeeded; virtual;
Persist the settings as a JSON file, named from LoadFromFile() parameter

property FileName: TFileName read fFileName;
Optional persistence file name, as set by LoadFromFile()

TRawUTF8ObjectCacheSettings = class(TSynPersistent)
Used by TRawUTF8ObjectCacheList to manage a list of information cache

constructor Create; override;
Will set default values to settings

property PurgePeriodMS: integer read fPurgePeriodMS write fPurgePeriodMS;
Period after which TRawUTF8ObjectCacheList will search for expired entries
- use -1 to disable purge (not advised, since may break process)
- default is 1000, i.e. 1 second
property TimeOutMS: integer read fTimeOutMS write fTimeOutMS;
   
   *Period after which the cache information should be flushed*
   - use -1 to disable time out; any big value will be limited to 10 minutes
   - default is 120000, i.e. 2 minutes

TRawUTF8ObjectCache = class(TSynAutoCreateFieldsLocked)
   
   *Maintain information cache for a given key*
   - after a given period of time, the entry is not deleted, but CacheClear virtual method is called to release the associated data or services
   - inherit from this abstract class to store your own key-defined information or your own interface-based services

   constructor Create(aOwner: TRawUTF8ObjectCacheList; const aKey: RawUTF8);
      reintroduce; virtual;
      
      *Initialize the information cache entry*
      - should not be called directly, but by TRawUTF8ObjectCacheList.GetLocked

destructor Destroy; override;
      
      *Finalize the information cache entry*
      - will also call the virtual CacheClear method

function Resolve(const aInterface: TGUID; out Obj): boolean;
   
   *Dependency Injection using fOwner.OnKeyResolve, for the current Key*

property Owner: TRawUTF8ObjectCacheList read fOwner;
   
   *Access to the associated storage list*

TRawUTF8ObjectCacheList = class(TRawUTF8List)
   
   *Manage a list of information cache, identified by a hashed key*
   - you should better inherit from this class, to give a custom name and constructor, or alter the default behavior
   - will maintain a list of TRawUTF8ObjectCache instances

   constructor Create(aClass: TRawUTF8ObjectCacheClass; aSettings: TRawUTF8ObjectCacheSettings; aLog: TSynLogFamily; aLogEvent: TSynLogInfo; const aOnKeyResolve: TOnKeyResolve); reintroduce;
      
      *Initialize the cache-information for a given class*
      - inherited classes may reintroduce a new constructor, for ease of use

destructor Destroy; override;
      
      *Finalize the cache information*

function GetLocked(const Key: RawUTF8; out cache: TRawUTF8ObjectCache; onlyexisting: boolean=false): boolean; virtual;
      
      *Fill TRawUTF8ObjectCache with the matching key information*
      - an unknown key, but with a successful NewObjectCache() call, will create and append a new fClass instance to the list (if onlyexisting is left to its default FALSE)
      - global or key-specific purge will be performed, if needed
      - on success (true), output cache instance will be locked
procedure AddToPurge(const Key: RawUTF8); virtual;

Register a key identifier so that next TryPurge will flush the entry
- a direct CacheClear may trigger a race condition in NewObjectCache: so you may use this function e.g. from a SOA callback

procedure ForceCacheClear;

This method will clear all associated information
- a regular Clear will destroy all TRawUTF8ObjectCache instances, whereas this method will call CacheClear on each entry, so will be more thread-safe and efficient in practice

procedure Log(const TextFmt: RawUTF8; const TextArgs: array of const; Level: TSynLogInfo = sllNone);

Access to the associated logging instance

procedure TryPurge;

You may call this method regularly to check for a needed purge
- if Settings.PurgePeriodMS is reached, each TRawUTF8ObjectCache instance will check for its TimeOutMS and call CacheClear if information is outdated

property OnKeyResolve: TOnKeyResolve read fOnKeyResolve write fOnKeyResolve;

Optional service locator for by-key Dependency Injection

TSynMonitorUsageID = object(TObject)

How the TSynMonitorUsage storage IDs are computed
- stored e.g. in TSQLMonitorUsage.ID primary key (after a shift)
- it follows a 23 bit pattern of hour (5 bit), day (5 bit), month (4 bit), year (9 bit - starting at 2016) so that it is monotonic over time
- by default, will store the information using mugHour granularity (i.e. values for the 60 minutes in a record), and pseudo-hours of 29, 30 and 31 (see USAGE_ID_HOURMARKER[]) will identify mugDay, mugMonth and mugYear consolidated statistics
- it will therefore store up to 24*365+365+12+1 = 9138 records per year in the associated storage engine (so there is no actual need to purge it)

Value: integer;

The TID, as computed from time and granularity

function GetTime(gran: TSynMonitorUsageGranularity; monthdaystartat0: boolean=false): integer;

Low-level read of a time field stored in this ID, per granularity

function Granularity: TSynMonitorUsageGranularity;

Retrieve the resolution of the stored information
- i.e. either mugHour, mugDay, mugMonth or mugYear, which will store a true 0..23 hour value (for mugHour), or 29/30/31 pseudo-hour (i.e. USAGE_ID_HOURMARKER[mugDay/mugMonth/mugYear])

function Text(Expanded: boolean; FirstTimeChar: AnsiChar = 'T'): RawUTF8;

Convert to ISO-8601 encoded text

function ToTimeLog: TTimeLog;

Returns the date/time
- minutes and seconds will set to 0
procedure From(Y,M,D: integer); overload;
    Computes an ID corresponding to mugDay granularity of a given time
    - hours, minutes and seconds will be merged
    - mugDay granularity will store 0..23 information about each hour
    - a pseudo hour of 29 (i.e. USAGE_ID_HOURMARKER[mugDay]) is used

procedure From(Y,M,D,H: integer); overload;
    Computes an ID corresponding to mugHour granularity of a given time
    - minutes and seconds will be ignored
    - mugHour granularity will store 0..59 information about each minute

procedure From(Y: integer); overload;
    Computes an ID corresponding to mugYear granularity of a given time
    - months, days, hours, minutes and seconds will be merged
    - mugYear granularity will store 0..11 information about each month
    - a pseudo hour of 31 (i.e. USAGE_ID_HOURMARKER[mugYear]) is used

procedure From(Y,M: integer); overload;
    Computes an ID corresponding to mugMonth granularity of a given time
    - days, hours, minutes and seconds will be merged
    - mugMonth granularity will store 0..31 information about each day
    - a pseudo hour of 30 (i.e. USAGE_ID_HOURMARKER[mugMonth]) is used

procedure FromNowUTC;
    Computes an ID corresponding to the current UTC date/time
    - minutes and seconds will be ignored

procedure FromTimeLog(const TimeLog: TTimeLog);
    Computes an ID corresponding to a given time
    - will set the ID with mugHour granularity, i.e. the information about the given hour, stored as
      per minute 0..59 values
    - minutes and seconds in supplied TimeLog value will therefore be ignored

procedure SetTime(gran: TSynMonitorUsageGranularity; aValue: integer);
    Low-level modification of a time field stored in this ID, per granularity

procedure Truncate(gran: TSynMonitorUsageGranularity);
    Change the resolution of the stored information

TSynMonitorUsage = class(TSynPersistentLock)
    Abstract class to track, compute and store TSynMonitor detailed statistics
    - you should inherit from this class to implement proper data persistence, e.g. using
      TSynMonitorUsageRest for ORM-based storage

destructor Destroy; override;
    Finalize the statistics, saving any pending information

function Modified(Instance: TObject; const PropNames: array of RawUTF8;
    ModificationTime: TTimeLog=0): integer; overload; virtual;
    To be called when tracked properties changed on a tracked class instance
### Function `Modified` (Instance: TObject):

```pascal
function Modified(Instance: TObject): integer; overload;
```

*To be called when tracked properties changed on a tracked class instance*

### Function `Track` (Instance: TObject; const Name: RawUTF8=''): integer; overload; virtual;

```pascal
function Track(Instance: TObject; const Name: RawUTF8=''): integer; overload;
```

*Track the values of one named object instance*
- will recognize the TSynMonitor* properties as TSynMonitorType from RTTI, using MonitorPropUsageValue(), within any (nested) object
- the instance will be stored in fTracked[].Instance: ensure it will stay available during the whole TSynMonitorUsage process

### Procedure `Track` (const Instances: array of TSynMonitor):

```pascal
procedure Track(const Instances: array of TSynMonitor); overload;
```

*Track the values of the given object instances*
- will recognize the TSynMonitor* properties as TSynMonitorType from RTTI, using MonitorPropUsageValue(), within any (nested) object
- instances will be stored in fTracked[].Instance: ensure they will stay available during the whole TSynMonitorUsage process

### Property `Comment`

```pascal
property Comment: RawUTF8 read fComment write fComment;
```

*Some custom text, associated with the current stored state*
- will be persistented by Save() methods

---

### Exception Classes

- **ESQLTableException** = `class(ESynException)`  
  *Exception raised in case of incorrect TSQLTable.Step / Field* use

- **EORMException** = `class(ESynException)`  
  *Generic parent class of all custom Exception types of this unit*

- **EORMBatchException** = `class(EORMException)`  
  *Exception raised in case of TSQLRestBatch problem*

- **EModelException** = `class(EORMException)`  
  *Exception raised in case of wrong Model definition*

- **EParsingException** = `class(EORMException)`  
  *Exception raised in case of unexpected parsing error*

- **ECommunicationException** = `class(EORMException)`  
  *Exception raised in case of a Client-Server communication error*

- **EBusinessLayerException** = `class(EORMException)`  
  *Exception raised in case of an error in project implementation logic*

- **ESecurityException** = `class(EORMException)`  
  *Exception raised in case of any authentication error*

- **EInterfaceFactoryException** = `class(ESynException)`  
  *Exception dedicated to interface factory, e.g. services and mock/stubs*

- **EInterfaceResolverException** = `class(ESynException)`  
  *Exception raised in case of Dependency Injection (aka IoC) issue*

- **EServiceException** = `class(EORMException)`  
  *Exception dedicated to interface based service implementation*
TSQLPropInfoRTTIInstance = class(TSQLPropInfoRTTIPtrInt)

*Information about a TSQLRecord class property*
- sftID for TSQLRecord properties, which are pointer(RecordID), not any true class instance
- sftMany for TSQLRecordMany properties, for which no data is stored in the table itself, but in a pivot table
- sftObject for e.g. TStrings TRawUTF8List TCollection instances

*constructor* Create(aPropInfo: PPropInfo; aPropIndex: integer; aSQLFieldType: TSQLFieldType; aOptions: TSQLPropInfoListOptions); **override**;

*Will setup the corresponding ObjectClass property*

*function* GetInstance(Instance: TObject): TObject;

*Direct access to the property class instance*

*procedure* SetInstance(Instance, Value: TObject);

*Direct access to the property class instance*

*property* ObjectClass: TClass read fObjectClass;

*Direct access to the property class*
- can be used e.g. for TSQLRecordMany properties

TSQLPropInfoRTTIRecordReference = class(TSQLPropInfoRTTIInt64)

*Information about a TRecordReference/TRecordReferenceToBeDeleted published property*
- identified as a sftRecord kind of property

*constructor* Create(aPropInfo: PPropInfo; aPropIndex: integer; aSQLFieldType: TSQLFieldType; aOptions: TSQLPropInfoListOptions); **override**;

*Will identify TRecordReferenceToBeDeleted kind of field, and setup the corresponding CascadeDelete property*

*property* CascadeDelete: boolean read fCascadeDelete;

*TRUE if this sftRecord is a TRecordReferenceToBeDeleted*

TSQLPropInfoRTTITID = class(TSQLPropInfoRTTIRecordReference)

*Information about a TID published property*
- identified as a sftTID kind of property, optionally tied to a TSQLRecord class, via its custom type name, e.g.

TSQLRecordClientID = type TID; -> TSQLRecordClient **class**

*constructor* Create(aPropInfo: PPropInfo; aPropIndex: integer; aSQLFieldType: TSQLFieldType; aOptions: TSQLPropInfoListOptions); **override**;

*Will setup the corresponding RecordClass property from the TID type name*
- the TSQLRecord type should have previously been registered to the TJSONSerializer.RegisterClassForJSON list, e.g. in TSQLModel.Create, so that e.g. 'TSQLRecordClientID' type name will match TSQLRecordClient
- in addition, the '...ToBeDeletedID' name pattern will set CascadeDelete
**property** CascadeDelete: boolean read fCascadeDelete;

TRUE if this sftTID type name follows the '...ToBeDeletedID' pattern
- e.g. 'TSQLRecordClientToBeDeletedID' type name will match TSQLRecordClient and set CascadeDelete
- is computed from its type name - for instance, if you define:

```pascal
type
  TSQLRecordClientToBeDeletedID = type TID;
  TSQLRecord = class(TSQLRecord)
    ...published
      OrderedBy: TSQLRecordClientToBeDeletedID read fOrderedBy write fOrderedBy;

...then this OrderedBy property will be tied to the TSQLRecordClient class of the corresponding model, and the whole record will be deleted when the targetting record is deleted (emulating a ON DELETE CASCADE)
```

**property** RecordClass: TSQLRecordClass read fRecordClass;

The TSQLRecord class associated to this TID
- is computed from its type name - for instance, if you define:

```pascal
type
  TSQLRecordClientID = type TID;
  TSQLRecord = class(TSQLRecord)
    ...published
      OrderedBy: TSQLRecordClientID read fOrderedBy write fOrderedBy;

...then this OrderedBy property will be tied to the TSQLRecordClient class of the corresponding model, and the field value will be reset to 0 when the targetting record is deleted (emulating a ON DELETE SET DEFAULT)
- equals TSQLRecord for plain TID field
- equals nil if T*ID type name doesn’t match any registered class
```

**TSQLPropInfoRTTIRecordVersion** = class(TSQLPropInfoRTTIInt64)

*Information about a TRecordVersion published property*
- identified as a sftRecordVersion kind of property, to track changes

**TSQLPropInfoRTTIID** = class(TSQLPropInfoRTTIInstance)

*Information about a TSQLRecord class TSQLRecord property*
- kind sftID, which are pointer(RecordID), not any true class instance
- will store the content just as an integer value
- will recognize any instance pre-allocated via Create*Joined() constructor

**procedure** GetJSONValues(Instance: TObject; W: TJSONSerializer); override;

*This method will recognize if the TSQLRecord was allocated by a Create*Joined() constructor: in this case, it will write the ID of the nested property, and not the PPtrInt() transtyped value*

**procedure** SetValue(Instance: TObject; Value: PUTF8Char; wasString: boolean); override;

*Raise an exception if was created by Create*Joined() constructor*

**TSQLPropInfoRTTIObject** = class(TSQLPropInfoRTTIInstance)

*Information about a TSQLRecord class TStrings/TRawUTF8List/TCollection property*
- kind sftObject e.g. for TStrings TRawUTF8List TCollection TObjectList instances
- binary serialization will store textual JSON serialization of the object, including custom serialization
**TSQLPropInfoRTTIMany** = class(TSQLPropInfoRTTIInstance)

Information about a TSQLRecord class TSQLRecordMany property
- kind sftMany, for which no data is stored in the table itself, but in a separated pivot table

**TSQLRecordProperties** = class(TObject)

Some information about a given TSQLRecord class properties
- used internally by TSQLRecord, via a global cache handled by this unit: you can access to each record's properties via TSQLRecord.RecordProps class
- such a global cache saves some memory for each TSQLRecord instance, and allows faster access to most wanted RTTI properties

*Used for DI-2.1.3 (page 2546).*

**ComputeBeforeAddFieldsBits**: TSQLFieldBits;
- Bit set to 1 for indicating TModTime/TCreateTime/TSessionUserID fields of this TSQLRecord
  - as applied before an INSERT
  - i.e. sftModTime, sftCreateTime and sftSessionUserID fields

**ComputeBeforeUpdateFieldsBits**: TSQLFieldBits;
- Bit set to 1 for indicating TModTime/TSessionUserID fields of this TSQLRecord (leaving TCreateTime untouched)
  - as applied before an UPDATE
  - i.e. sftModTime and sftSessionUserID fields

**CopiableFieldsBits**: TSQLFieldBits;
- Bit set to 1 for the all fields storing some data
  - match COPIABLE_FIELDS mask, i.e. all fields except sftMany

**FieldBits**: array[TSQLFieldType] of TSQLFieldBits;
- Bit set to 1 for indicating each TSQLFieldType fields of this TSQLRecord

**IsUniqueFieldsBits**: TSQLFieldBits;
- Bit set to 1 for an unique field
  - an unique field is defined as "stored AS_UNIQUE" (i.e. "stored false") in its property definition

**MainField**: array[boolean] of integer;
- Contains the main field index (e.g. mostly 'Name')
  - the [boolean] is for [ReturnFirstIfNoUnique] version
  - contains -1 if no field matches

**RTreeCoordBoundaryFields**: integer;
- Count of coordinate fields of a TSQLRecordRTree, before auxiliary columns

**SimpleFieldsBits**: array[TSQLOccasion] of TSQLFieldBits;
- Bit set to 1 for indicating fields to export, i.e. "simple" fields
  - this array will handle special cases, like the TCreateTime fields which shall not be included in soUpdate but solInsert and soSelect e.g.

**SimpleFieldsCount**: array[TSQLOccasion] of integer;
- Number of fields to export, i.e. "simple" fields
  - this array will handle special cases, like the TCreateTime fields which shall not be included in soUpdate but solInsert and soSelect e.g.
SmallFieldsBits: TSQLFieldBits;

* Bit set to 1 for the smallest simple fields
  - i.e. excluding non only sftBlob and sftMany, but also sftVariant, sftBlobDynArray, sftBlobCustom and sftUTF8Custom fields
  - may be used to minimize the transmitted content, e.g. when serializing to JSON for the most

**constructor** Create(aTable: TSQLRecordClass);

* Initialize the properties content

**destructor** Destroy; **override**;

* Release associated used memory

**function** AddFilterOrValidate(aFieldIndex: integer; aFilter: TSynFilterOrValidate): boolean; overload;

* Register a custom filter (transformation) or validation rule to the TSQMRecord class for a specified field
  - this will be used by TSQLRecord.Filter and TSQLRecord.Validate methods (in default implementation)
  - will return FALSE in case of an invalid field index

**function** AppendFieldName(FieldIndex: Integer; var Text: RawUTF8; ForceNoRowID: boolean): boolean;

* Append a field name to a RawUTF8 Text buffer
  - if FieldIndex=VIRTUAL_TABLE_ROWID_COLUMN (-1), appends 'RowID' or 'ID' (if ForceNoRowID=TRUE) to Text
  - on error (i.e. if FieldIndex is out of range) will return TRUE
  - otherwise, will return FALSE and append the field name to Text

**function** BlobFieldPropFromRawUTF8(const PropName: RawUTF8): PPropInfo;

* Retrieve a Field property RTTI information from a Property Name
  - this version returns nil if the property is not a BLOB field

**function** BlobFieldPropFromUTF8(PropName: PUTF8Char; PropNameLen: integer): PPropInfo;

* Retrieve a Field property RTTI information from a Property Name
  - this version returns nil if the property is not a BLOB field

**function** CheckBinaryHeader(var R: TFileBufferReader): boolean;

* Ensure that the TSQLRecord RTTI matches the supplied binary header
  - used e.g. by TSQLRestStorageInMemory.LoadFromBinary()

**function** CreateJSONWriter(JSON: TStream; Expand: boolean; const aFieldsCSV: RawUTF8; KnownRowsCount: integer; aBufSize: integer=8192): TJSONObjectSerializer;

* Create a TJSONWriter, ready to be filled with TSQLRecord.GetJSONValues()
  - this overloaded method will call FieldBitsFromCSV(aFieldsCSV,bits,withID) to retrieve the bits just like a SELECT (i.e. '*' for simple fields)

**function** CreateJSONWriter(JSON: TStream; Expand, withID: boolean; const aFields: TSQLFieldBits; KnownRowsCount: integer; aBufSize: integer=8192): TJSONObjectSerializer;

* Create a TJSONWriter, ready to be filled with TSQLRecord.GetJSONValues
  - you can use TSQLRecordProperties.FieldBitsFromCSV() or TSQLRecordProperties.FieldBitsFromRawUTF8() to compute aFields
function CreateJSONWriter(JSON: TStream; Expand, withID: boolean; const aFields: TSQLFieldIndexDynArray; KnownRowsCount: integer; aBufSize: integer=8192): TJSONSerializer; overload;

Create a TJSONWriter, ready to be filled with TSQLRecord.GetJSONValues(W)
- you can use TSQLRecordProperties.FieldBitsFromCSV() or
  TSQLRecordProperties.FieldBitsFromRawUTF8() to compute aFields

function CSVFromFieldBits(const Bits: TSQLFieldBits): RawUTF8;

Compute the CSV field names text from a set of bits

function FieldBitsFromBlobField(aBlobField: PPropInfo; var Bits: TSQLFieldBits): boolean;

Set all bits corresponding to the supplied BLOB field type information
- returns TRUE on success, FALSE if blob field is not recognized

function FieldBitsFromCSV(const aFieldsCSV: RawUTF8; var Bits: TSQLFieldBits): boolean; overload;

Set all bits corresponding to the supplied CSV field names
- returns TRUE on success, FALSE if any field name is not existing

function FieldBitsFromCSV(const aFieldsCSV: RawUTF8; var Bits: TSQLFieldBits; out withID: boolean): boolean; overload;

Set all bits corresponding to the supplied CSV field names, including ID
- returns TRUE on success, FALSE if any field name is not existing
- this overloaded method will identify ID/RowID field name, and set withID output parameter according to its presence
- if aFieldsCSV='*', Bits will contain all simple fields, and withID=true

function FieldBitsFromCSV(const aFieldsCSV: RawUTF8): TSQLFieldBits; overload;

Set all bits corresponding to the supplied CSV field names
- returns the matching fields set

function FieldBitsFromExcludingCSV(const aFieldsCSV: RawUTF8; aOccasion: TSQLOccasion=soSelect): TSQLFieldBits;

Set all simple bits corresponding to the simple fields, excluding some
- could be a convenient alternative to FieldBitsFromCSV() if only some fields are to be excluded
- returns the matching fields set

function FieldBitsFromRawUTF8(const aFields: array of RawUTF8; var Bits: TSQLFieldBits): boolean; overload;

Set all bits corresponding to the supplied field names
- returns TRUE on success, FALSE if any field name is not existing

function FieldBitsFromRawUTF8(const aFields: array of RawUTF8): TSQLFieldBits; overload;

Set all bits corresponding to the supplied field names
- returns the matching fields set

function FieldIndexDynArrayFromBlobField(aBlobField: PPropInfo; var Indexes: TSQLFieldIndexDynArray): boolean;

Set all field indexes corresponding to the supplied BLOB field type information
- returns TRUE on success, FALSE if blob field is not recognized
function FieldIndexDynArrayFromCSV(const aFieldsCSV: RawUTF8; var Indexes: TSQLFieldIndexDynArray): boolean; overload;
   Set all field indexes corresponding to the supplied CSV field names
   - returns TRUE on success, FALSE if any field name is not existing

function FieldIndexDynArrayFromCSV(const aFieldsCSV: RawUTF8): TSQLFieldIndexDynArray; overload;
   Set all field indexes corresponding to the supplied CSV field names
   - returns the matching fields set

function FieldIndexDynArrayFromRawUTF8(const aFields: array of RawUTF8; var Indexes: TSQLFieldIndexDynArray): boolean; overload;
   Set all field indexes corresponding to the supplied field names
   - returns TRUE on success, FALSE if any field name is not existing

function FieldIndexDynArrayFromRawUTF8(const aFields: array of RawUTF8): TSQLFieldIndexDynArray; overload;
   Set all field indexes corresponding to the supplied field names
   - returns the matching fields set

function IsFieldName(const PropName: RawUTF8): boolean;
   Return TRUE if the given name is either ID/RowID, either a property name

function IsFieldNameOrFunction(const PropName: RawUTF8): boolean;
   Return TRUE if the given name is either ID/RowID, either a property name, or an aggregate function (MAX/MIN/AVG/SUM) on a valid property name

function MainFieldName(ReturnFirstIfNoUnique: boolean=false): RawUTF8;
   Return the first unique property of kind RawUTF8
   - this property is mainly the "Name" property, i.e. the one with "stored AS_UNIQUE" (i.e. "stored false") definition on most TSQLRecord
   - if ReturnFirstIfNoUnique is TRUE and no unique property is found, the first RawUTF8 property is returned anyway
   - returns " if no matching field was found

function SaveSimpleFieldsFromJsonArray(var P: PUTF8Char; var EndOfObject: AnsiChar; ExtendedJSON: boolean): RawUTF8;
   Convert a JSON array of simple field values into a matching JSON object

function SetCustomCollation(const aFieldName, aCollationName: RawUTF8): boolean; overload;
   Set a custom SQlite3 text column collation for a specified field
   - overloaded method which expects the field to be named

function SetCustomCollation(FieldIndex: integer; const aCollationName: RawUTF8): boolean; overload;
   Set a custom SQlite3 text column collation for a specified field
   - can be used e.g. to override the default COLLATE SYSTEMNOCASE of RawUTF8
   - collations defined within our SynSQLite3 unit are named BINARY, NOCASE, RTRIM and our custom SYSTEMNOCASE, ISO8601, WIN32CASE, WIN32NOCASE
   - do nothing if FieldIndex is not valid, and returns false
   - could be set in overridden class procedure TSQLRecord.InternalDefineModel so that it will be common to all database models, for both client and server
function SQLAddField(FieldIndex: integer): RawUTF8;

Return the UTF-8 encoded SQL statement source to alter the table for adding the specified field

function SQLFieldTypeToSQL(FieldIndex: integer): RawUTF8;

Return the SQLite3 field datatype for each specified field
- set to " for fields with no column created in the database (e.g. sftMany)
- returns e.g. 'INTEGER,' or 'TEXT COLLATE SYSTEMNOCASE,'

procedure AddFilterOrValidate(const aFieldName: RawUTF8; aFilter: TSynFilterOrValidate); overload;

Register a custom filter (transformation) or validation to the TSQLRecord class for a specified field
- this will be used by TSQLRecord.Filter and TSQLRecord.Validate methods (in default implementation)
- will raise an EModelException if the field name does not exist

procedure RegisterCustomFixedSizeRecordProperty(aTable: TClass; aRecordSize: cardinal; const aName: RawUTF8; aPropertyPointer: pointer; aAttributes: TSQLPropInfoAttributes; aFieldWidth: integer; aData2Text: TOnSQLPropInfoRecord2Text=nil; aText2Data: TOnSQLPropInfoRecord2Data=nil);

Add a custom unmanaged fixed-size record property
- simple kind of records (i.e. those not containing reference-counted members) do not have RTTI generated, at least in older versions of Delphi
- use this method within TSQLRecord.InternalRegisterCustomProperties overridden method to define a custom record property with no reference-counted types within (like strings) - typical use may be TGUID
- main parameters are the record size, in bytes, and the property pointer
- add an TSQLPropInfoRecordFixedSize instance to the internal list
- if aData2Text/aText2Data parameters are not defined, it will fallback to TSQLPropInfo.BinaryToText() simple text Base64 encoding
- can be used to override the default TSQLRecord corresponding method:

class procedure TSQLMyRecord.InternalRegisterCustomProperties(Props: TSQLRecordProperties);

begin
  Props.RegisterCustomFixedSizeRecordProperty(self,SizeOf(TMyRec),'RecField',
    @TSQLMyRecord(nil).fRecField, [], SizeOf(TMyRec));
end;
procedure RegisterCustomPropertyFromRTTI(aTable: TClass; aTypeInfo: PTypeInfo; const aName: RawUTF8; aPropertyPointer: pointer; aAttributes: TSQLPropInfoAttributes=[]; aFieldWidth: integer=0);

Add a custom property from its RTTI definition stored as JSON
- handle any kind of record with TypeInfo() generated
- use this method within InternalRegisterCustomProperties overridden method to define a custom record property containing reference-counted types
- main parameters are the record RTTI information, and the property pointer
- add an TSQLPropInfoCustomJSON instance to the internal list
- can be used as such:
  class procedure TSQLMyRecord.InternalRegisterCustomProperties( Props: TSQLRecordProperties);
  begin
    Props.RegisterCustomPropertyFromRTTI(self,TypeInfo(TMyRec), 'RecField', @TSQLMyRecord(nil).fRecField);
  end;

procedure RegisterCustomPropertyFromTypeName(aTable: TClass; const aTypeName, aName: RawUTF8; aPropertyPointer: pointer; aAttributes: TSQLPropInfoAttributes=[]; aFieldWidth: integer=0);

Add a custom property from its type name, stored as JSON
- handle any kind of registered record, including TGUID
- use this method within InternalRegisterCustomProperties overridden method to define a custom record property containing reference-counted types
- main parameters are the record RTTI information, and the property pointer
- add an TSQLPropInfoCustomJSON instance to the internal list
- can be used as such:
  class procedure TSQLMyRecord.InternalRegisterCustomProperties( Props: TSQLRecordProperties);
  begin
    Props.RegisterCustomPropertyFromTypeName(self, 'TGUID', 'GUID', @TSQLMyRecord(nil).fGUID,[aIsUnique],38);
  end;

procedure RegisterCustomRTTIRecordProperty(aTable: TClass; aRecordInfo: PTypeInfo; const aName: RawUTF8; aPropertyPointer: pointer; aAttributes: TSQLPropInfoAttributes=[]; aFieldWidth: integer=0; aData2Text: TOnSQLPropInfoRecord2Text=nil; aText2Data: TOnSQLPropInfoRecord2Data=nil);

Add a custom record property from its RTTI definition
- handle any kind of record with TypeInfo() generated
- use this method within InternalRegisterCustomProperties overridden method to define a custom record property containing reference-counted types
- main parameters are the record RTTI information, and the property pointer
- add an TSQLPropInfoRecordRTTI instance to the internal list
- can be used as such:
  class procedure TSQLMyRecord.InternalRegisterCustomProperties( Props: TSQLRecordProperties);
  begin
    Props.RegisterCustomRTTIRecordProperty(self,TypeInfo(TMyRec), 'RecField', @TSQLMyRecord(nil).fRecField);
  end;
**procedure** SaveBinaryHeader(W: TFileBufferWriter);

Save the TSQLRecord RTTI into a binary header
- used e.g. by TSQLRestStorageInMemory.SaveToBinary()

**procedure** SetCustomCollationForAll(aFieldType: TSQLFieldType; const aCollationName: RawUTF8);

Set a custom SQLite3 text column collation for a given field type
- can be used e.g. to override ALL default COLLATE SYSTEMNOCASE of RawUTF8, or the default COLLATE ISO8601 of TDateTime, and let the generated SQLite3 file be available outside the scope of mORMot's SQLite3 engine
- collations defined within our SynSQLite3 unit are named BINARY, NOCASE, RTRIM and our custom SYSTEMNOCASE, ISO8601, WIN32CASE, WIN32NOCASE
- could be set in overridden class procedure TSQLRecord.InternalDefineModel so that it will be common to all database models, for both client and server
- note that you may inherit from TSQLRecordNoCase to use the NOCASE standard SQLite3 collation for all descendant ORM objects

**procedure** SetJSONWriterColumnNames(W: TJSONSerializer; KnownRowsCount: integer);

Set the W.ColNames[] array content + W.AddColumns

**procedure** SetMaxLengthFilterForTextFieldFields(IndexIsUTF8Length: boolean=false);

Allow to filter the length of all text published properties of this table
- the "index" attribute of the RawUTF8/string published properties could be used to specify a maximum length for external VARCHAR() columns
- SQLite3 will just ignore this "index" information, but it could be handy to be able to filter the value length before sending to the DB
- this method will create TSynFilterTruncate corresponding to the maximum field size specified by the "index" attribute, to filter before write
- will expect the "index" value to be in UTF-16 codepoints, unless IndexIsUTF8Length is set to TRUE, indicating UTF-8 length in "index"

**procedure** SetMaxLengthValidatorForTextFieldFields(IndexIsUTF8Length: boolean=false);

Allow to validate length of all text published properties of this table
- the "index" attribute of the RawUTF8/string published properties could be used to specify a maximum length for external VARCHAR() columns
- SQLite3 will just ignore this "index" information, but it could be handy to be able to validate the value length before sending to the DB
- this method will create TSynValidateText corresponding to the maximum field size specified by the "index" attribute, to validate before write
- will expect the "index" value to be in UTF-16 codepoints, unless IndexIsUTF8Length is set to TRUE, indicating UTF-8 length in "index"

**procedure** SetVariantFieldsDocVariantOptions(const Options: TDocVariantOptions);

Customize the TDocVariant options for all variant published properties
- will change the TSQLPropInfoRTTIVariant.DocVariantOptions value
- use e.g. as SetVariantFieldDocVariantOptions(JSON_OPTIONS_FAST_EXTENDED)
- see also TSQLRecordNoCaseExtended root class

**property** BlobCustomFields: TSQLPropInfoObjArray read fBlobCustomFields;

List of all sftBlobCustom fields of this TSQLRecord
- have been defined e.g. as TSQLPropInfoCustom custom definition
property BlobFields: TSQLPropInfoRTTIObjArray read fBlobFields;
List all BLOB fields of this TSQLRecord
- i.e. generic sftBlob fields (not sftBlobDynArray, sftBlobCustom nor sftBlobRecord)

property CopiableFields: TSQLPropInfoObjArray read fCopiableFields;
List all fields which can be copied from one TSQLRecord instance to another
- match COPIABLE_FIELDS mask, i.e. all fields except sftMany

property DynArrayFields: TSQLPropInfoRTTIDynArrayObjArray read fDynArrayFields;
List of all sftBlobDynArray fields of this TSQLRecord

property DynArrayFieldsHasObjArray: boolean read fDynArrayFieldsHasObjArray;
TRUE if any of the sftBlobDynArray fields of this TSQLRecord is a T*ObjArray
- used e.g. by TSQLRecord.Destroy to release all owned nested instances

property Fields: TSQLPropInfoList read fFields;
List all fields, as retrieved from RTTI

property Filters: TSynFilterOrValidateObjArrayArray read fFilters;
All TSynFilter or TSynValidate instances registered per each field
- since validation and filtering are used within some CPU-consuming part of the framework (like UI edition), both filters and validation rules are grouped in the same list - for TSynTableFieldProperties there are separated Filters[] and Validates[] arrays, for better performance

property HasNotSimpleFields: boolean read fHasNotSimpleFields;
If this class has any BLOB or TSQLRecodMany fields
- i.e. some fields to be ignored

property HasTypeFields: TSQLFieldTypes read fHasTypeFields;
Set of field types appearing in this record

property JoinedFields: TSQLPropInfoRTTIIDObjArray read fJoinedFields;
List all TSQLRecord fields of this TSQLRecord
- ready to be used by TSQLTableJSON.CreateFromTables()
- i.e. the class itself then, all fields of type sftID (excluding sftMany)

property JoinedFieldsTable: TSQLRecordClassDynArray read fJoinedFieldsTable;
Wrapper of all nested TSQLRecord class of this TSQLRecord
- ready to be used by TSQLTableJSON.CreateFromTables()
- i.e. the class itself as JoinedFieldsTable[0], then, all nested TSQLRecord published properties (of type sftID, ergo excluding sftMany)
- equals nil if there is no nested TSQLRecord property (i.e. JoinedFields=nil)

property ManyFields: TSQLPropInfoRTTIManyObjArray read fManyFields;
List all TSQLRecordMany fields of this TSQLRecord

property RecordManyDestProp: TSQLPropInfoRTTIInstance read fRecordManyDestProp;
For a TSQLRecordMany class, points to the Dest property RTTI

property RecordManySourceProp: TSQLPropInfoRTTIInstance read fRecordManySourceProp;
For a TSQLRecordMany class, points to the Source property RTTI
property RecordVersionField: TSQLPropInfoRTTIRecordVersion read fRecordVersionField;
  Points to any TRecordVersion field
  - contains nil if no such stfRecordVersion field do exist
  - will be used by low-level storage engine to compute and store the monotonic version number
during any write operation

property SimpleFields: TSQLPropInfoObjArray read fSimpleFields;
  List all "simple" fields of this TSQLRecord
  - by default, the TSQLRawBlob and TSQLRecordMany fields are not included into this set: they
    must be read specifically (in order to spare bandwidth for BLOBs)
  - dynamic arrays belong to simple fields: they are sent with other properties content
  - match inverted NOT_SIMPLE_FIELDS mask

property SQLTableName: RawUTF8 read fSQLTableName;
  The Table name in the database, associated with this TSQLRecord class
  - 'TSQL' or 'TSQLRecord' chars are trimmed at the beginning of the ClassName
  - or the ClassName is returned as is, if no 'TSQL' or 'TSQLRecord' at first

property SQLTableNameUpperWithDot: RawUTF8 read fSQLTableNameUpperWithDot;
  The Table name in the database in uppercase with a final '.'
  - e.g. 'TEST.' for TSQLRecordTest class
  - can be used with IdemPChar() for fast check of a table name

property SQLTableRetrieveAllFields: RawUTF8 read fSQLTableRetrieveAllFields;
  Returns 'COL1,COL2' with all COL* set to all field names, including RowID, TRecordVersion and
  BLOBs
  - this won't change depending on the ORM settings: so it can be safely computed here and not
    in TSQLModelRecordProperties
  - used e.g. by TSQLRest.InternalListJSON()

property SQLTableRetrieveBlobFields: RawUTF8 read fSQLTableRetrieveBlobFields;
  Returns 'COL1,COL2' with all BLOB columns names
  - used e.g. by TSQLRestServerDB.RetrieveBlobFields()

property SQLTableSimpleFieldsNoRowID: RawUTF8 read fSQLTableSimpleFieldsNoRowID;
  Returns 'COL1,COL2' with all COL* set to simple field names
  - same value as SQLTableSimpleFields[false,false]
  - this won't change depending on the ORM settings: so it can be safely computed here and not
    in TSQLModelRecordProperties
  - used e.g. by TSQLRecord.GetSQLValues

property SQLTableUpdateBlobFields: RawUTF8 read fSQLTableUpdateBlobFields;
  Returns 'COL1=?,COL2=?' with all BLOB columns names
  - used e.g. by TSQLRestServerDB.UpdateBlobFields()

property Table: TSQLRecordClass read fTable;
  The TSQLRecord class

property TableClassProp: PClassProp read fClassProp;
  Fast access to the RTTI properties attribute
property TableClassType: PClassType read fClassType;
  Fast access to the RTTI properties attribute

TSQLRestURIParams = object(TObject)
  Store all parameters for a Client or Server method call
  - as used by TSQLRestServer.URI or TSQLRestClientURI.InternalURI

  InBody: RawUTF8;
  Input parameter containing the caller message body
  - e.g. some GET/POST/PUT JSON data can be specified here

  InHead: RawUTF8;
  Input parameter containing the caller message headers
  - you can use e.g. to retrieve the remote IP:
    Call.Header(HEADER_REMOTEIP_UPPER)
    or FindNameValue(Call.InHead,HEADER_REMOTEIP_UPPER)
    but consider rather using TSQLRestServerURIContext.RemoteIP

  LowLevelConnectionID: Int64;
  Opaque reference to the protocol context which made this request
  - may point e.g. to a THttpServerResp, a TWebSocketServerResp, a THttpApiServer, a
    TSQLRestClientURI, a TFastCGIServer or a TSQLRestServerNamedPipeResponse instance
  - stores SynCrtSock's THttpServerConnectionID, i.e. a Int64 as expected by http.sys, or an
    incremental rolling sequence of 31-bit integers for THttpServer/TWebSocketServer, or maybe a
    raw PInt(self/THandle)

  LowLevelFlags: TSQLRestURIParamsLowLevelFlags;
  Low-level properties of the current protocol context

  Method: RawUTF8;
  Input parameter containing the caller method
  - handle enhanced REST codes: LOCK/UNLOCK/BEGIN/END/ABORT

  OutBody: RawUTF8;
  Output parameter to be set to the response message body

  OutHead: RawUTF8;
  Output parameter to be set to the response message header
  - it is the right place to set the returned message body content type, e.g.
  TEXT_CONTENT_TYPE_HEADER or HTTP_CONTENT_TYPE_HEADER: if not set, the default
  JSON_CONTENT_TYPE_HEADER will be returned to the client, meaning that the message is JSON
  - you can use OutBodyType() function to retrieve the stored content-type

  OutInternalState: cardinal;
  Output parameter to be set to the database internal state

  OutStatus: cardinal;
  Output parameter to be set to the HTTP status integer code
  - HTTP_NOTFOUND=404 e.g. if the url doesn't start with Model.Root (caller can try another
    TSQLRestServer)
RestAccessRights: PSQLAccessRights;

Associated RESTful access rights
- AccessRights must be handled by the TSQlRestServer child, according to the Application
  Security Policy (user logging, authentication and rights management) - making access rights a
  parameter allows this method to be handled as pure stateless, thread-safe and session-free

Url: RawUTF8;

Input parameter containing the caller URI

function Header(UpperName: PAnsiChar): RawUTF8;

Just a wrapper around FindNameValue(InHead,UpperName)
- use e.g. as
  Call.Header(HEADER_REMOTEIP_UPPER) or Call.Header(HEADER_BEARER_UPPER)
- consider rather using TSQlRestServerURIContext.InHeader[] or even dedicated
  TSQlRestServerURIContext.RemoteIP/AuthenticationBearerToken

function HeaderOnce(var Store: RawUTF8; UpperName: PAnsiChar): RawUTF8;

Wrap FindNameValue(InHead,UpperName) with a cache store

function InBodyType(GuessJSONIfNoneSet: boolean=True): RawUTF8;

Retrieve the "Content-Type" value from InHead
- if GuessJSONIfNoneSet is TRUE, returns JSON if none was set in headers

function InBodyTypeIsJson(GuessJSONIfNoneSet: boolean=True): boolean;

Check if the "Content-Type" value from InHead is JSON
- if GuessJSONIfNoneSet is TRUE, assume JSON is used

function OutBodyType(GuessJSONIfNoneSet: boolean=True): RawUTF8;

Retrieve the "Content-Type" value from OutHead
- if GuessJSONIfNoneSet is TRUE, returns JSON if none was set in headers

function OutBodyTypeIsJson(GuessJSONIfNoneSet: boolean=True): boolean;

Check if the "Content-Type" value from OutHead is JSON
- if GuessJSONIfNoneSet is TRUE, assume JSON is used

procedure Init; overload;

Initialize the non RawUTF8 values

procedure Init(const aURI,aMethod,aInHead,aInBody: RawUTF8); overload;

Initialize the input values

TServiceRunningContext = record

Will identify the currently running service on the server side
- is the type of the global ServiceContext thread
  - to access the current TSQlRestServer instance (and e.g. its ORM/CRUD or SOA methods), use
    Request.Server and not Factory.Server, which may not be available e.g. if you run the service from
    the server side (so no factory is involved)
- note that the safest (and slightly faster) access to the TSQlRestServer instance associated with a
  service is to inherit your implementation class from TInjectableObjectRest
Factory: TServiceFactoryServer;

The currently running service factory
- it can be used within server-side implementation to retrieve the associated TSQLRestServer instance
- note that TServiceFactoryServer.Get() won't override this value, when called within another service (i.e. if Factory is not nil)

Request: TSQLRestServerURIContext;

The currently running context which launched the method
- low-level RESTful context is also available in its Call member
- Request.Server is the safe access point to the underlying TSQLRestServer, unless the service is implemented via TInjectableObjectRest, so the TInjectableObjectRest.Server property is preferred
- make available e.g. current session or authentication parameters (including e.g. user details via Request.Server.SessionGetUser)

RunningThread: TThread;

The thread which launched the request
- is set by TSQLRestServer.BeginCurrentThread from multi-thread server handlers - e.g. TSQLite3HttpServer or TSQLRestServerNamedPipeResponse

TServiceFactoryExecution = record

Internal per-method list of execution context as hold in TServiceFactory

Denied: set of 0..255;

The list of denied TSQLAuthGroup ID(s)
- used on server side within TSQLRestServerURIContext.ExecuteSOAByInterface
- bit 0 for client TSQLAuthGroup.ID=1 and so on...
- is therefore able to store IDs up to 256
- void by default, i.e. no denial = all groups allowed for this method

LogClass: TSQLRecordServiceLogClass;

The TSQLRecordServiceLog class to use, as defined in LogRest.Model

LogRest: TSQLRest;

Where execution information should be written as TSQLRecordServiceLog

Options: TServiceMethodOptions;

Execution options for this method (about thread safety or logging)

TSQLAccessRights = object(TObject)

Set the User Access Rights, for each Table
- one property for every and each URI method (GET/POST/PUT/DELETE)
- one bit for every and each Table in Model.Tables[]

AllowRemoteExecute: TSQLAllowRemoteExecute;

Set of allowed actions on the server side

DELETE: TSQLFieldTables;

DELETE method (delete record) table access bits
GET: TSQLFieldTables;

GET method (retrieve record) table access bits
- note that a GET request with a SQL statement without a table (i.e. on 'ModelRoot' URI with a SQL statement as SentData, as used in TSQLRestClientURI.UpdateFromServer) will be checked for simple cases (i.e. the first table in the FROM clause), otherwise will follow , whatever the bits here are: since TSQLRestClientURI.UpdateFromServer() is called only for refreshing a direct statement, it will be OK; you can improve this by overriding the TSQLRestServer.URI() method
- if the REST request is LOCK, the PUT access bits will be read instead of the GET bits value

POST: TSQLFieldTables;

POST method (create record) table access bits

PUT: TSQLFieldTables;

PUT method (update record) table access bits
- if the REST request is LOCK, the PUT access bits will be read instead of the GET bits value

function CanExecuteORMWrite(Method: TSQLURIMethod; Table: TSQLRecordClass; TableIndex: integer; const TableID: TID; Context: TSQLRestServerURIContext): boolean;

Validate mPost/mPut/mDelete action against those access rights
- used by TSQLRestServerURIContext.ExecuteORMWrite and TSQLRestServer.EngineBatchSend methods for proper security checks

function ToString: RawUTF8;

Serialize the content as TEXT
- use the TSQLAuthGroup.AccessRights CSV format

procedure Edit(aModel: TSQLModel; aTable: TSQLRecordClass; aRights: TSQLOccasions); overload;

Wrapper method which can be used to set the CRUD abilities over a table
- will raise an EModelException if the supplied table is incorrect
- use TSQLOccasion set as parameter

procedure Edit(aTableIndex: integer; C, R, U, D: Boolean); overload;

Wrapper method which can be used to set the CRUD abilities over a table
- C=Create, R=Read, U=Update, D=Delete rights

procedure Edit(aTableIndex: integer; aRights: TSQLOccasions); overload;

Wrapper method which can be used to set the CRUD abilities over a table
- use TSQLOccasion set as parameter

procedure Edit(aModel: TSQLModel; aTable: TSQLRecordClass; C, R, U, D: Boolean); overload;

Wrapper method which can be used to set the CRUD abilities over a table
- will raise an EModelException if the supplied table is incorrect
- C=Create, R=Read, U=Update, D=Delete rights

procedure FromString(P: PUTF8Char);

Unserialize the content from TEXT
- use the TSQLAuthGroup.AccessRights CSV format
TSQLRestServerURIContext = class(TObject)

Abstract calling context for a TSQLRestServerCallback event handler
- having a dedicated class avoid changing the implementation methods signature if the framework
  add some parameters or behavior to it
- see TSQLRestServerCallback for general code use
- most of the internal methods are declared as virtual, so it allows any kind of custom routing or
  execution scheme
- instantiated by the TSQLRestServer.URI() method using its ServicesRouting property
- see TSQLRestRoutingREST and TSQLRestRoutingJSON_RPC for working inherited classes - NEVER
  set this abstract TSQLRestServerURIContext class to TSQLRest.ServicesRouting property!

Call: PSQLRestURIParams;

Access to all input/output parameters at TSQLRestServer.URI() level
- process should better call Results() or Success() methods to set the appropriate answer or
  Error() method in case of an error
- low-level access to the call parameters can be made via this pointer

Command: TSQLRestServerURIContextCommand;

The current execution command

CustomErrorMsg: RawUTF8;

Optional error message which will be transmitted as JSON error (if set)
- contains e.g. TNotifyAuthenticationFailedReason text during
  TSQLRestServer.OnAuthenticationFailed event call, or the reason of a
  TSQLRestServer.RecordCanBeUpdated failure

ForceServiceResultAsJSONObject: boolean;

Force the interface-based service methods to return a JSON object
- default behavior is to follow Service.ResultAsJSONObject property value (which own default is
  to return a more convenient JSON array)
- if set to TRUE, this execution context will FORCE the method to return a JSON object, even if
  Service.ResultAsJSONObject=false: this may be handy when the method is executed from a
  JavaScript content

ForceServiceResultAsJSONObjectWithoutResult: boolean;

Force the interface-based service methods to return a plain JSON object
- i.e. '{....}' instead of '{"result":{....]}'
- only set if ForceServiceResultAsJSONObject=TRUE and if no ID is about to be returned
- could be used e.g. for stateless interaction with a (non mORMot) stateless JSON REST Server

ForceServiceResultAsXMLObject: boolean;

Force the interface-based service methods to return a XML object
- default behavior is to follow Service.ResultAsJSONObject property value (which own default is
  to return a more convenient JSON array)
- if set to TRUE, this execution context will FORCE the method to return a XML object, by setting
  ForceServiceResultAsJSONObject then converting the resulting JSON object into the
  corresponding XML via JSONBufferToXML()
- TSQLRestServerURIContext.InternalExecuteSOAByInterface will inspect the Accept HTTP
  header to check if the answer should be XML rather than JSON
Synopsense (Now known as Synopsense) is a software framework for developing database-driven applications. It is designed to be modular, scalable, and easy to use. This document provides a detailed description of the framework's architecture and features.

### mORMot Framework

**Software Architect**

**Design 1.18**

**Date:** September 16, 2020

---

#### ForceServiceResultAsXMLObjectNameSpace: RawUTF8;

- Specify a custom name space content when returning a XML object
- Default behavior is to follow Service.ResultAsXMLObjectNameSpace property (which is void by default)
- Service may set e.g. XMLUTF8_NAMESPACE, which will append `<content ...> </content>` around the generated XML data, to avoid validation problems or set a particular XML name space, depending on the application

#### JWTContent: TJWTContent;

- JWT validation information, as filled by AuthenticationCheck()

#### Log: TSynLog;

- Associated logging instance for the current thread on the server
- You can use it to log some process on the server side

#### Method: TSQLURIMethod;

- The used Client-Server method (matching the corresponding HTTP Verb)
- This property will be set from incoming URI, even if RESTful authentication is not enabled

#### MethodIndex: integer;

- The index of the callback published method within the internal class list

#### MicroSecondsElapsed: QWord;

- High-resolution timing of the execution command, in micro-seconds
- Only set when TSQLRestServer.URI finished

#### Parameters: PUTF8Char;

- URI inlined parameters
  - Use UrlDecodeValue*() functions to retrieve the values
  - For mPOST requests, will also be filled for following content types: application/x-www-form-urlencoded or multipart/form-data

#### ParametersPos: integer;

- URI inlined parameters position in Call^.url string
  - Use Parameters field to retrieve the values

#### SafeProtocolID: integer;

- The internal ID used to identify modelroot/_safe_ custom encryption

#### Server: TSQLRestServer;

- The associated TSQLRestServer instance which executes its URI method

#### Service: TServiceFactoryServer;

- The service identified by an interface-based URI

#### ServiceExecution: PServiceFactoryExecution;

- The current execution context of an interface-based service
  - Maps to Service.fExecution[ServiceMethodIndex-SERVICE_PSEUDO_METHOD_COUNT]

#### ServiceExecutionOptions: TServiceMethodOptions;

- The current execution options of an interface-based service
  - Contain ServiceExecution.Options including optNoLogInput/optNoLogOutput in case of TInterfaceFactory.RegisterUnsafeSPIType
ServiceInstanceID: PtrUInt;

The instance ID for interface-based services instance
- can be e.g. the client session ID for sicPerSession or the thread ID for sicPerThread

ServiceMethod: pointer;

Access to the raw PServiceMethod information of an interface-based URI
- equals nil if ServiceMethodIndex in 0..2 (pseudo-methods)

ServiceMethodIndex: integer;

The method index for an interface-based service
- Service member has already be retrieved from URI (so is not nil)
- 0..2 are the internal _free_/_contract_/_signature_ pseudo-methods

ServiceParameters: PUTF8Char;

The JSON array of parameters for an interface-based service
- Service member has already be retrieved from URI (so is not nil)

Session: cardinal;

The corresponding session TAuthSession.IDCardinal value
- equals 0 (CONST_AUTHENTICATION_SESSION_NOT_STARTED) if the session is not started yet - i.e. if still in handshaking phase
- equals 1 (CONST_AUTHENTICATION_NOT_USED) if authentication mode is not enabled - i.e. if TSQLRestServer.HandleAuthentication = FALSE

SessionGroup: integer;

The corresponding TAuthSession.User.GroupRights.ID value
- is undefined if Session is 0 or 1 (no authentication running)

SessionUser: TID;

The corresponding TAuthSession.User.ID value
- is undefined if Session is 0 or 1 (no authentication running)

SessionUserName: RawUTF8;

The corresponding TAuthSession.User.LogonName value
- is undefined if Session is 0 or 1 (no authentication running)

Static: TSQLRest;

The static instance corresponding to the associated Table (if any)

StaticKind: TSQLRestServerKind;

The kind of static instance corresponding to the associated Table (if any)

Table: TSQLRecordClass;

The Table as specified at the URI level (if any)

TableEngine: TSQLRest;

The RESTful instance implementing the Table specified at the URI level (if any)
- equals TSQLRestServer most of the time, but may be an TSQLRestStorage for any in-memory/MongoDB/virtual instance

TableID: TID;

The associated TSQLRecord.ID, as decoded from URI scheme
- this property will be set from incoming URI, even if RESTful authentication is not enabled
TableIndex: integer;
   The index in the Model of the Table specified at the URI level (if any)

TableRecordProps: TSQLModelRecordProperties;
   The RTTI properties of the Table specified at the URI level (if any)

URI: RawUTF8;
   The URI address, excluding trailing /info and ?par1=.... parameters
   - can be either the table name (in RESTful protocol), or a service name

URIAfterRoot: PUTF8Char;
   Points inside Call^.URI, after the 'root/' prefix

URIBlobFieldName: RawUTF8;
   The optional Blob field name as specified in URI
   - e.g. retrieved from "ModelRoot/TableName/TableID/BlobFieldName"

URISessionSignaturePos: integer;
   Position of the &session_signature=... text in Call^.url string

URIWithoutSignature: RawUTF8;
   Same as Call^.URI, but without the &session_signature=... ending

constructor Create(aServer: TSQLRestServer; const aCall: TSQLRestURIPrams);
   virtual;

   Initialize the execution context
   - this method could have been declared as protected, since it should never be called outside the
     TSQLRestServer.URI() method workflow
   - should set Call, and Method members

destructor Destroy; override;

   Finalize the execution context

function AuthenticationBearerToken: RawUTF8;

   Retrieve the "Authorization: Bearer <token>" value from incoming HTTP headers
   - typically returns a JWT for stateless self-contained authentication, as expected by
     TJWTAbstract.Verify method
   - as an alternative, a non-standard and slightly less safe way of token transmission may be to
     encode its value as ?authenticationbearer=.... URI parameter (may be convenient when
     embedding resources in HTML DOM)

function AuthenticationCheck(jwt: TJWTAbstract): boolean; virtual;

   Validate "Authorization: Bearer <JWT>" content from incoming HTTP headers
   - returns true on success, storing the payload in the JWTContent field
   - set JWTContent.result = jwtNoToken if jwt is nil
   - on failure (i.e. returns false), will set the error context as 403 HTTP_FORBIDDEN so that you
     may directly write:

   procedure TMyDaemon.Files(Ctxt: TSQLRestServerURIContext);
   begin
      if Ctxt.AuthenticationCheck(fJWT) then
         Ctxt.ReturnFileFromFolder(‘c:\datafolder’);
   end;
function ClientKind: TSQLRestServerURIContextClientKind;
Identify which kind of client is actually connected
- the "User-Agent" HTTP will be checked for 'mORMot' substring, and set ckFramework on match
- either ckAjax for a classic (AJAX) browser, or any other kind of HTTP client
- will be used e.g. by ClientSQLRecordOptions to check if the current remote client expects standard JSON in all cases

function ClientSQLRecordOptions: TJSONSerializerSQLRecordOptions;
Identify if the request is about a Table containing nested objects or arrays, which could be serialized as JSON objects or arrays, instead of plain JSON string (as stored in the database)
- will identify ClientKind=ckAjax, or check for rsogetAsJsonNotAsString in TSQLRestServer.Options

function GetInputAsTDocVariant(const Options: TDocVariantOptions; ServiceMethod: pointer): variant;
Retrieve all input parameters from URI as a variant JSON object
- returns Unassigned if no parameter was defined
- returns a JSON object with input parameters encoded as
  
  "name1":value1,"name2":value2...

- optionally with a PServiceMethod information about the actual values types
- if the parameters were encoded as multipart, the JSON object will be encoded with its textual values, or with nested objects, if the data was supplied as binary:
  
  "name1":{"data":..."filename":..."contenttype":...},"name2":...

since name1.data will be Base64 encoded, so you should better use the InputAsMultiPart() method instead when working with binary

function InputAsMultiPart(var MultiPart: TMultiPartDynArray): Boolean;
Decode any multipart/form-data POST request input
- returns TRUE and set MultiPart array as expected, on success

function InputEnum(const ParamName: RawUTF8; EnumType: PTypeInfo; out ValueEnum; DefaultEnumOrd: integer=0): boolean;
Retrieve one input parameter from its URI name as an enumeration
- will expect the value to be specified as integer, or as the textual representation of the enumerate, ignoring any optional lowercase prefix as featured by TEnumType.GetEnumNameValue()
- returns TRUE and set ValueEnum if the parameter is found and correct
- returns FALSE and set ValueEnum to first item (i.e. DefaultEnumOrd) if the parameter is not found, or not containing a correct value

function InputOrError(const ParamName: RawUTF8; out Value: variant; const ErrorMessageForMissingParameter: string): boolean;
Retrieve one input parameter from its URI name as variant
- returns FALSE and call Error(ErrorMessageForMissingParameter) - which may be a resourcestring - if the parameter is not found
- returns TRUE and set Value if the parameter is found
- if the parameter value is text, it is stored in the variant as a RawUTF8: so before Delphi 2009, you won't loose any Unicode character, but you should convert its value to AnsiString using UTF8ToString()
function InputUTF8OrDefault(const ParamName, DefaultValue: RawUTF8): RawUTF8;

Retrieve one input parameter from its URI name as RawUTF8
- returns supplied DefaultValue if the parameter is not found

function InputUTF8OrError(const ParamName: RawUTF8; out Value: RawUTF8; const ErrorMessageForMissingParameter: string): boolean;

Retrieve one input parameter from its URI name as RawUTF8
- returns FALSE and call Error(ErrorMessageForMissingParameter) - which may be a resourcestring - if the parameter is not found
- returns TRUE and set Value if the parameter is found

function IsRemoteAdministrationExecute: boolean;
True if called from TSQLRestServer.AdministrationExecute

class procedure ClientSideInvoke(var uri: RawUTF8; ctxt: TSQLRestServerURIContextClientInvoke; const method, params, clientDrivenID: RawUTF8; out sent,head: RawUTF8); virtual; abstract;

At Client Side, compute URI and BODY according to the routing scheme
- abstract implementation which is to be overridden
- as input, method should be the method name to be executed, params should contain the incoming parameters as JSON CSV (without []), and clientDriven ID should contain the optional Client ID value
- at output, should update the HTTP uri corresponding to the proper routing, and should return the corresponding HTTP body within sent

procedure ConfigurationRestMethod(SettingsStorage: TObject);
Implements a method-based service for live update of some settings
- should be called from a method-based service, e.g. Configuration()
- the settings are expected to be stored e.g. in a TSynAutoCreateFields instance, potentially with nested objects
- accept the following REST methods to read and write the settings:
  GET http://server:888/root/configuration
  GET http://server:888/root/configuration/propname
  GET http://server:888/root/configuration/propname?value=propvalue
- could be used e.g. as such:
  procedure TMyRestServerMethods.Configuration(Ctxt: TSQLRestServerURIContext);
  begin
    http://server:888/myrestserver/configuration/name?value=newValue
  end;

procedure Error(const Format: RawUTF8; const Args: array of const; Status: integer=HTTP_BADREQUEST; CacheControlMaxAge: integer=0); overload;

Use this method to send back an error to the caller
- implementation is just a wrapper over Error(FormatUTF8(Format,Args))

procedure Error(const ErrorMessage: RawUTF8=''; Status: integer=HTTP_BADREQUEST; CacheControlMaxAge: integer=0); overload; virtual;

Use this method to send back an error to the caller
- expects Status to not be HTTP_SUCCESS neither HTTP_CREATED, and will send back a JSON error message to the caller, with the supplied error text
- set CacheControlMaxAge<>0 to include a Cache-Control: max-age=xxx header
- if no ErrorMessage is specified, will return a default text corresponding to the Status code
procedure Error(E: Exception; const Format: RawUTF8; const Args: array of const; Status: integer=HTTP_BADREQUEST); overload;

*Use this method to send back an error to the caller*
- will serialize the supplied exception, with an optional error message

procedure FillInput(const LogInputIdent: RawUTF8='');

*Extract the input parameters from its URI*
- you should not have to call this method directly, but rather all the InputInt/InputDouble/InputUTF8/InputExists/... properties
- may be useful if you want to access directly to InputPairs[] with no prior knowledge of the input parameter names
- you can specify a title text to optionally log the input array

procedure Redirect(const NewLocation: RawUTF8; PermanentChange: boolean=false);

*Use this method notify the caller that the resource URI has changed*
- returns a HTTP_TEMPORARYREDIRECT status with the specified location, or HTTP_MOVEDPERMANENTLY if PermanentChange is TRUE

procedure Results(const Values: array of const; Status: integer=HTTP_SUCCESS; Handle304NotModified: boolean=false; CacheControlMaxAge: integer=0);

*Use this method to send back a JSON object with a "result" field*
- this method will encode the supplied values as a "result":"..." JSON object, as such for one value:
  
  ```
  "result":"OneValue"
  ```

  (with one value, you can just call TSQLRestClientURI.CallBackGetResult method to call and decode this value) or as a JSON object containing an array of values:
  
  ```
  "result":
  "One","two"
  ```

  - expects Status to be either HTTP_SUCCESS or HTTP_CREATED
  - caller can set Handle304NotModified=TRUE for Status=HTTP_SUCCESS and/or set CacheControlMaxAge<>0 to include a Cache-Control: max-age=xxx header

procedure ReturnBlob(const Blob: RawByteString; Status: integer=HTTP_SUCCESS; Handle304NotModified: boolean=true; const FileName: TFileName=''; CacheControlMaxAge: integer=0);

*Uses this method to send back directly any binary content to the caller*
- the exact MIME type will be retrieved using GetMimeContentTypeHeader(), from the supplied Blob binary buffer, and optional a file name
- by default, the HTTP_NOTMODIFIED process will take place, to minimize bandwidth between the server and the client
- set CacheControlMaxAge<>0 to include a Cache-Control: max-age=xxx header
procedure ReturnFile(const FileName: TFileName; Handle304NotModified: boolean=false; const ContentType: RawUTF8=''; const AttachmentFileName: RawUTF8=''; const Error404Redirect: RawUTF8=''; CacheControlMaxAge: integer=0);

*Use this method to send back a file to the caller*
- this method will let the HTTP server return the file content
- if Handle304NotModified is TRUE, will check the file age to ensure that the file content will be sent back to the server only if it changed; set CacheControlMaxAge<>0 to include a Cache-Control: max-age=xxx header
- if ContentType is left to default '', method will guess the expected mime-type from the file name extension
- if the file name does not exist, a generic 404 error page will be returned, unless an explicit redirection is defined in Error404Redirect
- you can also specify the resulting file name, as downloaded and written by the client browser, in the optional AttachmentFileName parameter, if the URI does not match the expected file name

procedure ReturnFileFromFolder(const FolderName: TFileName; Handle304NotModified: boolean=true; const DefaultFileName: TFileName='index.html'; const Error404Redirect: RawUTF8=''; CacheControlMaxAge: integer=0);

*Use this method to send back a file from a local folder to the caller*
- URIBlobFieldName value, as parsed from the URI, will contain the expected file name in the local folder, using DefaultFileName if the URI is void, and redirecting to Error404Redirect if the file is not found
- this method will let the HTTP server return the file content
- if Handle304NotModified is TRUE, will check the file age to ensure that the file content will be sent back to the server only if it changed set CacheControlMaxAge<>0 to include a Cache-Control: max-age=xxx header

procedure Returns(const NameValuePairs: array of const; Status: integer=HTTP_SUCCESS; Handle304NotModified: boolean=false; HandleErrorAsRegularResult: boolean=false; const CustomHeader: RawUTF8=''); overload;

*Use this method to send back a JSON object to the caller*
- this method will encode the supplied values e.g. as
  ```
  JSONEncode(['name','John','year',1972]) = '{"name":"John","year":1972}'
  ```
- implementation is just a wrapper around Returns(JSONEncode([]))
- note that cardinal values should be type-casted to Int64() (otherwise the integer mapped value will be transmitted, therefore wrongly)
- expects Status to be either HTTP_SUCCESS or HTTP_CREATED
- caller can set Handle304NotModified=TRUE for Status=HTTP_SUCCESS
procedure Returns(const Result: RawUTF8; Status: integer=HTTP_SUCCESS; const CustomHeader: RawUTF8=''; Handle304NotModified: boolean=false; HandleErrorAsRegularResult: boolean=false; CacheControlMaxAge: integer=0; ServerHash: RawUTF8=''); overload;

   Use this method to send back directly a result value to the caller
   - expects Status to be either HTTP_SUCCESS, HTTP_NOTMODIFIED, HTTP_CREATED, or HTTP_TEMPORARYREDIRECT, and will return as answer the supplied Result content with no transformation
   - if Status is an error code, it will call Error() method
   - CustomHeader optional parameter can be set e.g. to TEXT_CONTENT_TYPE_HEADER if the default JSON_CONTENT_TYPE is not OK, or calling GetMimeContentTypeHeader() on the returned binary buffer
   - if Handle304NotModified is TRUE and Status is HTTP_SUCCESS, the Result content will be hashed (using crc32c) and in case of no modification will return HTTP_NOTMODIFIED to the browser, without the actual result content (to save bandwidth)
   - set CacheControlMaxAge<>0 to include a Cache-Control: max-age=xxx header

procedure Returns(Value: TObject; Status: integer=HTTP_SUCCESS; Handle304NotModified: boolean=false; SQLRecordOptions: TJSONSerializerSQLRecordOptions=[]; const CustomHeader: RawUTF8=''); overload;

   Use this method to send back any object as JSON document to the caller
   - this method will call ObjectToJson() to compute the returned content
   - you can customize SQLRecordOptions, to force the returned JSON object to have its TSQLRecord nested fields serialized as true JSON arrays or objects, or add an "ID_str" string field for JavaScript

procedure ReturnsJson(const Value: variant; Status: integer=HTTP_SUCCESS; Handle304NotModified: boolean=false; Escape: TTextWriterKind=twJSONEscape; MakeHumanReadable: boolean=false; const CustomHeader: RawUTF8='');

   Use this method to send back any variant as JSON to the caller
   - this method will call VariantSaveJSON() to compute the returned content

procedure Success(Status: integer=HTTP_SUCCESS); virtual;

   Use this method if the caller expect no data, just a status
   - just wrap the overloaded Returns() method with no result value
   - if Status is an error code, it will call Error() method
   - by default, calling this method will mark process as successful

property InCookie[CookieName: RawUTF8]: RawUTF8 read GetInCookie write SetInCookie;
   Retrieve an incoming HTTP cookie value
   - cookie name are case-sensitive

property InHeader[const HeaderName: RawUTF8]: RawUTF8 read GetInHeader;
   Retrieve an incoming HTTP header
   - the supplied header name is case-insensitive
   - but rather call RemoteIP or UserAgent properties instead of InHeader['remoteip'] or InHeader['User-Agent']

property Input[const ParamName: RawUTF8]: variant read GetInput;
   Retrieve one input parameter from its URI name as variant
   - if the parameter value is text, it is stored in the variant as a generic VCL string content: so before Delphi 2009, you may lose some characters at decoding from UTF-8 input buffer
   - raise an EParsingException if the parameter is not found
property InputDouble[const ParamName: RawUTF8]: double read GetInputDouble;
  Retrieve one input parameter from its URI name as double
  - raise an EParsingException if the parameter is not found

property InputDoubleOrVoid[const ParamName: RawUTF8]: double read GetInputDoubleOrVoid;
  Retrieve one input parameter from its URI name as double
  - returns 0 if the parameter is not found

property InputExists[const ParamName: RawUTF8]: Boolean read GetInputExists;
  Return TRUE if the input parameter is available at URI
  - even if InputUTF8['param']='', there may be '..?param=&another=2'

property InputHexaOrVoid[const ParamName: RawUTF8]: cardinal read GetInputHexaOrVoid;
  Retrieve one hexadecimal input parameter from its URI name as cardinal
  - returns 0 if the parameter is not found

property InputInt[const ParamName: RawUTF8]: Int64 read GetInputInt;
  Retrieve one input parameter from its URI name as Int64
  - raise an EParsingException if the parameter is not found

property InputIntOrVoid[const ParamName: RawUTF8]: Int64 read GetInputIntOrVoid;
  Retrieve one input parameter from its URI name as Int64
  - returns 0 if the parameter is not found

property InputOrVoid[const ParamName: RawUTF8]: variant read GetInputOrVoid;
  Retrieve one input parameter from its URI name as variant
  - if the parameter value is text, it is stored in the variant as a RawUTF8: so before Delphi 2009,
    you won't lose any Unicode character, but you should convert its value to AnsiString using
    UTF8ToString()
  - returns Unassigned if the parameter is not found

property InputPairs: TRawUTF8DynArray read FInput;
  Low-level access to the input parameters, stored as pairs of UTF-8
  - even items are parameter names, odd are values
  - Input[*] properties should have been called previously to fill the internal array, or by calling
    FillInput if you do not know the input parameters which may appear

property InputString[const ParamName: RawUTF8]: string read GetInputString;
  Retrieve one input parameter from its URI name as a VCL string
  - raise an EParsingException if the parameter is not found
  - prior to Delphi 2009, some Unicode characters may be missing in the returned AnsiString value

property InputStringOrVoid[const ParamName: RawUTF8]: string read GetInputStringOrVoid;
  Retrieve one input parameter from its URI name as a VCL string
  - returns "" if the parameter is not found
  - prior to Delphi 2009, some Unicode characters may be missing in the returned AnsiString value

property InputUTF8[const ParamName: RawUTF8]: RawUTF8 read GetInputUTF8;
  Retrieve one input parameter from its URI name as RawUTF8
  - raise an EParsingException if the parameter is not found
property InputUTF8OrVoid[const ParamName: RawUTF8]: RawUTF8 read
GetInputUTF8OrVoid;

Retrieve one input parameter from its URI name as RawUTF8
- returns "" if the parameter is not found

property OutSetCookie: RawUTF8 read fOutSetCookie write SetOutSetCookie;

Define a new 'name=value' cookie to be returned to the client
- if not void, TSQlRestServer.URI() will define a new 'set-cookie: ...' header in Call^.OutHead
- you can use COOKIE_EXPIRED as value to delete a cookie in the browser
- if no Path=/.. is included, it will append
  '; Path=' + Server.Model.Root + '; HttpOnly'

property RemoteIP: RawUTF8 read GetRemoteIP;

Retrieve the "RemoteIP" value from the incoming HTTP headers

property RemoteIPIsLocalHost: boolean read GetRemoteIPIsLocalHost;

True if the "RemoteIP" value from the incoming HTTP headers is '127.0.0.1'

property RemoteIPNotLocal: RawUTF8 read GetRemoteIPNotLocal;

"RemoteIP" value from the incoming HTTP headers but "" for '127.0.0.1'

property ResourceFileName: TFileName read GetResourceFileName;

Compute the file name corresponding to the URI
- e.g. '/root/methodname/toto/index.html' will return 'toto\index.html'

property UserAgent: RawUTF8 read GetUserAgent;

Retrieve the "User-Agent" value from the incoming HTTP headers

TSQlRestRoutingREST = class(TSQLRestServerURIContext)

Calling context for a TSQLRestServerCallBack using simple REST for interface-based services
- this class will use RESTful routing for interface-based services: method name will be identified
  within the URI, as
  /Model/Interface.Method[/ClientDrivenID]
  e.g. for ICalculator.Add:
  POST /root/Calculator.Add
  \(...\)
  [1,2]
  or, for a sicClientDriven mode service:
  POST /root/ComplexNumber.Add/1234
  \(...\)
  [20,30]

in this case, the sent content will be a JSON array of [parameters...]
- as an alternative, input parameters may be encoded at URI level (with a size limit depending on
  the HTTP routers, whereas there is no such limitation when they are transmitted as message
  body)
- one benefit of having .../ClientDrivenID encoded at URI is that it will be more secured in our
  RESTful authentication scheme: each method and even client driven session will be signed
  individualy
At Client Side, compute URI and BODY according to RESTful routing scheme
- e.g. on input uri='root/Calculator', method='Add', params='1,2' and clientDrivenID='1234' -> on output uri='root/Calculator.Add/1234' and sent='[1,2]'

Calling context for a TSQLRestServerCallBack using JSON/RPC for interface-based services
- in this routing scheme, the URI will define the interface, then the method name will be inlined with parameters, e.g.
  POST /root/Calculator
  (...)  
  
  or, for a sicClientDriven mode service:
  POST /root/ComplexNumber
  (...)  
  

At Client Side, compute URI and BODY according to JSON/RPC routing scheme
- e.g. on input uri='root/Calculator', method='Add', params='1,2' and clientDrivenID='1234' -> on output uri='root/Calculator' and sent='"method":"Add","params":[1,2],"id":1234'

Description of a method-based service

Set to TRUE disable Authentication check for this method
- use TSQLRestServer.ServiceMethodByPassAuthentication() method

The event which will be executed for this method

Detailed statistics associated with this method

Internal data used by TSQLRecord.FillPrepare()/FillPrepareMany() methods
- using a dedicated class will reduce memory usage for each TSQLRecord instance (which won't need these properties most of the time)

Finalize the mapping
function Fill(aRow: integer; aDest: TSQLRecord): Boolean; overload;
  Fill a TSQLRecord published properties from a TSQLTable row
  - overloaded method using a specified destination record to be filled
  - won't work with cross-reference mapping (FillPrepareMany)
  - use the mapping prepared with Map() method

function Fill(aRow: integer): Boolean; overload;
  Fill a TSQLRecord published properties from a TSQLTable row
  - use the mapping prepared with Map() method

function TableMapFields: TSQLFieldBits;
  Return all mapped fields, or [[] if nil

procedure ComputeSetUpdatedFieldBits(Props: TSQLRecordProperties; out Bits: TSQLFieldBits);
  Used to compute the updated field bits during a fill
  - will return Props.SimpleFieldsBits[soUpdate] if no fill is in process

procedure Fill(aTableRow: PPUtf8CharArray; aDest: TSQLRecord); overload;
  Fill a TSQLRecord published properties from a TSQLTable row
  - overloaded method using a specified destination record to be filled
  - won't work with cross-reference mapping (FillPrepareMany)
  - use the mapping prepared with Map() method
  - aTableRow will point to the first column of the matching row

procedure Fill(aTableRow: PPUtf8CharArray); overload;
  Fill a TSQLRecord published properties from a TSQLTable row
  - use the mapping prepared with Map() method
  - aTableRow will point to the first column of the matching row

procedure Map(aRecord: TSQLRecord; aTable: TSQLTable; aCheckTableName: TSQLCheckTableName);
  Map all columns of a TSQLTable to a record mapping

procedure UnMap;
  Reset the mapping
  - is called e.g. by TSQLRecord.FillClose
  - will free any previous Table if necessary
  - will release TSQLRecordMany.Dest instances as set by TSQLRecord.FillPrepareMany()

property FillCurrentRow: integer read fFillCurrentRow;
  The current Row during a Loop

property JoinedFields: boolean read GetJoinedFields;
  Equals TRUE if the instance was initialized via TSQLRecord.CreateJoined()
  TSQLRecord.CreateAndFillPrepareJoined()
  - it means that all nested TSQLRecord are pre-allocated instances, not trans-typed pointer(IDs)

property Table: TSQLTable read fTable;
  The TSQLTable stated as FillPrepare() parameter
  - the internal temporary table is stored here for TSQLRecordMany
  - this instance is freed by TSQLRecord.Destroy if fTable.OwnerMustFree=true
TSQLRestBatch = class(TObject)

Used to store a BATCH sequence of writing operations
- is used by TSQLRest to process BATCH requests using BatchSend() method, or TSQLRestClientURI
  for its Batch*() methods
- but you can create your own stand-alone BATCH process, so that it will be able to make some
  transactional process - aka the "Unit Of Work" pattern

constructor Create(aRest: TSQLRest; aTable: TSQLRecordClass;
AutomaticTransactionPerRow: cardinal=0; Options: TSQLRestBatchOptions=[];
InternalBufferSize: cardinal=65536); virtual;

Begin a BATCH sequence to speed up huge database changes
- each call to normal Add/Update/Delete methods will create a Server request, therefore can be
  slow (e.g. if the remote server has bad ping timing)
- start a BATCH sequence using this method, then call BatchAdd() BatchUpdate() or
  BatchDelete() methods to make some changes to the database
- when BatchSend will be called, all the sequence transactions will be sent as one to the remote
  server, i.e. in one URI request
- if BatchAbort is called instead, all pending BatchAdd/Update/Delete transactions will be
  aborted, i.e. ignored
- expect one TSQLRecordClass as parameter, which will be used for the whole sequence (in this
  case, you can't mix classes in the same BATCH sequence)
- if no TSQLRecordClass is supplied, the BATCH sequence will allow any kind of individual record
  in BatchAdd/BatchUpdate/BatchDelete
- return TRUE on success, FALSE if aTable is incorrect or a previous BATCH sequence was already
  initiated
- should normally be used inside a Transaction block: there is no automated
  TransactionBegin..Commit/RollBack generated in the BATCH sequence if you leave the default
  AutomaticTransactionPerRow=0 parameter - but this may be a concern with a lot of concurrent
  clients
- you should better set AutomaticTransactionPerRow > 0 to execute all BATCH processes within
  an unique transaction grouped by a given number of rows, on the server side - set
  AutomaticTransactionPerRow=maxInt if you want one huge transaction, or set a convenient
  value (e.g. 10000) depending on the back-end database engine abilities, if you want to retain the
  transaction log file small enough for the database engine
- BatchOptions could be set to tune the SQL execution, e.g. force INSERT OR IGNORE on internal
  SQLite3 engine
- InternalBufferSize could be set to some high value (e.g. 10 shl 20), if you expect a very high
  number of rows in this BATCH

destructor Destroy; override;

Finalize the BATCH instance
function Add(Value: TSQLRecord; SendData: boolean; ForceID: boolean=false; const CustomFields: TSQLFieldBits=[]; DoNotAutoComputeFields: boolean=false): integer;

Create a new member in current BATCH sequence
- work in BATCH mode: nothing is sent to the server until BatchSend call
- returns the corresponding index in the current BATCH sequence, -1 on error
- if SendData is true, content of Value is sent to the server as JSON
- if ForceID is true, client sends the Value.ID field to use this ID for adding the record (instead of a database-generated ID)
- if Value is TSQLRecordFTS3/4/5, Value.ID is stored to the virtual table
- Value class MUST match the TSQLRecordClass used at BatchStart, or may be of any kind if no class was specified
- BLOB fields are NEVER transmitted here, even if ForceBlobTransfer=TRUE
- if CustomFields is left void, the simple fields will be used; otherwise, you can specify your own set of fields to be transmitted when SendData=TRUE (including BLOBs, even if they will be Base64-encoded within JSON content) - CustomFields could be computed by TSQLRecordProperties.FieldBitsFromCSV() or TSQLRecordProperties.FieldBitsFromRawUTF8(), or by setting ALL_FIELDS
- this method will always compute and send TCreateTime/TModTime fields

function Delete(ID: TID): integer; overload;

Delete a member in current BATCH sequence
- work in BATCH mode: nothing is sent to the server until BatchSend call
- returns the corresponding index in the current BATCH sequence, -1 on error
- deleted record class is the TSQLRecordClass used at BatchStart() call: it will fail if no class was specified for this BATCH sequence

function Delete(Table: TSQLRecordClass; ID: TID): integer; overload;

Delete a member in current BATCH sequence
- work in BATCH mode: nothing is sent to the server until BatchSend call
- returns the corresponding index in the current BATCH sequence, -1 on error
- with this overloaded method, the deleted record class is specified: no TSQLRecordClass shall have been set at BatchStart() call

function PrepareForSending(out Data: RawUTF8): boolean; virtual;

Close a BATCH sequence started by Start method
- Data is ready to be supplied to TSQLRest.BatchSend() overloaded method
- will also notify the TSQLRest.Cache for all deleted IDs
- you should not have to call it in normal use cases

function RawAdd(const SentData: RawUTF8): integer;

Allow to append some JSON content to the internal raw buffer
- could be used to emulate Add() with an already pre-computed JSON object
- returns the corresponding index in the current BATCH sequence, -1 on error

function RawAppend(FullRow: boolean=true): TTextWriter;

Allow to append some JSON content to the internal raw buffer
- could be used to emulate Add/Update/Delete
- FullRow=TRUE will increment the global Count
function RawUpdate(const SentData: RawUTF8; ID: TID): integer;

Allow to append some JSON content to the internal raw buffer for a PUT
- could be used to emulate Update() with an already pre-computed JSON object
- returns the corresponding index in the current BATCH sequence, -1 on error

function Update(Value: TSQLRecord; const CustomFields: TSQLFieldBits=[];
DoNotAutoComputeFields: boolean=false; ForceCacheUpdate: boolean=false): integer;
overload;

Update a member in current BATCH sequence
- work in BATCH mode: nothing is sent to the server until BatchSend call
- returns the corresponding index in the current BATCH sequence, -1 on error
- Value class MUST match the TSQLRecordClass used at BatchStart, or may be of any kind if no
class was specified
- BLOB fields are NEVER transmitted here, even if ForceBlobTransfert=TRUE
- if Value has an opened FillPrepare() mapping, only the mapped fields will be updated (and also
ID and TModTime fields) - FillPrepareMany() is not handled yet (all simple fields will be updated)
- if CustomFields is left void, the simple fields will be used, or the fields retrieved via a previous
FillPrepare() call; otherwise, you can specify your own set of fields to be transmitted (including
BLOBs, even if they will be Base64-encoded within the JSON content) - CustomFields could be
computed by TSQLRecordProperties.FieldBitsFromCSV() or
TSQLRecordProperties.FieldBitsFromRawUTF8()
- this method will always compute and send any TModTime fields, unless
DoNotAutoComputeFields is set to true
- if not all fields are specified, will reset the cache entry associated with this value, unless
ForceCacheUpdate is TRUE

function Update(Value: TSQLRecord; const CustomCSVFields: RawUTF8;
DoNotAutoComputeFields: boolean=false; ForceCacheUpdate: boolean=false): integer;
overload;

Update a member in current BATCH sequence
- work in BATCH mode: nothing is sent to the server until BatchSend call
- is an overloaded method to Update(Value,FieldBitsFromCSV())

procedure Reset(aTable: TSQLRecordClass; AutomaticTransactionPerRow: cardinal=0;
Options: TSQLRestBatchOptions=[]); overload; virtual;

Reset the BATCH sequence so that you can re-use the same TSQLRestBatch

procedure Reset; overload;

Reset the BATCH sequence to its previous state
- could be used to prepare a next chunk of values, after a call to TSQLRest.BatchSend

property AddCount: integer read fAddCount;

How many times Add() has been called for this BATCH process

property Count: integer read GetCount;

Retrieve the current number of pending transactions in the BATCH sequence

property DeleteCount: integer read fDeleteCount;

How many times Delete() has been called for this BATCH process

property OnWrite: TOnBatchWrite read fOnWrite write fOnWrite;

This event handler will be triggered by each Add/Update/Delete method
property Rest: TSQLRest read fRest;
   Read only access to the associated TSQLRest instance

property SizeBytes: cardinal read GetSizeBytes;
   Retrieve the current JSON size of pending transaction in the BATCH sequence

property Table: TSQLRecordClass read fTable;
   Read only access to the main associated TSQLRecord class (if any)

property UpdateCount: integer read fUpdateCount;
   How many times Update() has been called for this BATCH process

TSQLRestBatchLocked = class(TSQLRestBatch)
   Thread-safe class to store a BATCH sequence of writing operations

constructor Create(aRest: TSQLRest; aTable: TSQLRecordClass;
   AutomaticTransactionPerRow: cardinal=0; Options: TSQLRestBatchOptions=[];
   InternalBufferSize: cardinal=65536); override;
   Initialize the BATCH instance

destructor Destroy; override;
   Finalize the BATCH instance

procedure Reset(aTable: TSQLRecordClass; AutomaticTransactionPerRow: cardinal=0;
   Options: TSQLRestBatchOptions=[]); override;
   Reset the BATCH sequence so that you can re-use the same TSQLRestBatch

property ResetTix: Int64 read fTix write fTix;
   Property set to the current GetTickCount64 value when Reset is called

property Safe: TSynLocker read fSafe;
   Access to the locking methods of this instance
   - use Safe.Lock/TryLock with a try ... finally Safe.Unlock block

property Threshold: integer read fThreshold write fThreshold;
   May be used to store a number of rows to flush the content
**TSQLRecord = class(TObject)**

**Root class for defining and mapping database records**
- inherits a class from TSQLRecord, and add published properties to describe the table columns (see TPropInfo for SQL and Delphi type mapping/conversion)
- this published properties can be auto-filled from TSQLTable answer with FillPrepare() and FillRow(), or FillFrom() with TSQLTable or JSON data
- these published properties can be converted back into UTF-8 encoded SQL source with GetSQLValues or GetSQLSet or into JSON format with GetJSONValues
- BLOB fields are decoded to auto-freeing TSQLRawBlob properties
- any published property defined as a T*ObjArray dynamic array storage of persistents (via TJSONSerializer.RegisterObjArrayForJSON) will be freed
- consider inherit from TSQLRecordNoCase and TSQLRecordNoCaseExtended if you expect regular NOCASE collation and smaller (but not standard JSON) variant fields persistence

*Used for DI-2.1.1 (page 2543), DI-2.1.2 (page 2545), DI-2.1.3 (page 2546).*

**constructor** Create(aClient: TSQLRest; aID: TID; ForUpdate: boolean=false); overload;

*This constructor initializes the object as above, and fills its content from a client or server connection*
- if ForUpdate is true, the REST method is LOCK and not GET: it tries to lock the corresponding record, then retrieve its content; caller has to call UnLock() method after Value usage, to release the record

**constructor** Create(const aSimpleFields: array of const; aID: TID); overload;

*This constructor initializes the record and set the simple fields with the supplied values*
- the aSimpleFields parameters must follow explicitely the order of published properties of the aTable class, excepting the TSQLRawBlob and TSQLRecordMany kind (i.e. only so called "simple fields") - in particular, parent properties must appear first in the list
- the aSimpleFields must have exactly the same count of parameters as there are "simple fields" in the published properties
- will raise an EORMException in case of wrong supplied values

**constructor** Create; overload; virtual;

*This constructor initializes the record*
- auto-instantiate any TSQLRecordMany instance defined in published properties
- override this method if you want to use some internal objects (e.g. TStringList or TCollection as published property)
**constructor** Create(aClient: TSQLRest; const FormatSQLWhere: RawUTF8; const ParamsSQLWhere, BoundsSQLWhere: array of const); overload;

This constructor initializes the object as above, and fills its content from a client or server connection, using a specified WHERE clause with parameters
- the FormatSQLWhere clause will replace all '%' chars with the supplied ParamsSQLWhere[] values, and all '?' chars with BoundsSQLWhere[] values, as :(...) : Inlined parameters - you should either call:
  ```pascal
  Rec := TSQLRecord.Create(aClient, 'Count=(%):'[aCount],[]);
  or (letting the inlined parameters being computed by FormatUTF8)
  Rec := TSQLRecord.Create(aClient, 'Count=?',[],[aCount]);
  or even better, using the other Create overloaded constructor:
  Rec := TSQLRecord.Create(aClient, 'Count=?',[aCount]);
  - using '?' and BoundsSQLWhere[] is perhaps more readable in your code, and will in all case create a request with :(...) : inline parameters, with automatic RawUTF8 quoting if necessary
```

**constructor** Create(aClient: TSQLRest; aPublishedRecord: TSQLRecord; ForUpdate: boolean=false); overload;

This constructor initializes the object and fills its content from a client or server connection, from a TSQLRecord published property content
- is just a wrapper around Create(aClient,PtrInt(aPublishedRecord)) or Create(aClient, aPublishedRecord.ID)
- a published TSQLRecord property is not a class instance, but a typecast to TObject(RecordID) - you can also use its ID property
- if ForUpdate is true, the REST method is LOCK and not GET: it tries to lock the corresponding record, then retrieve its content; caller has to call UnLock() method after Value usage, to release the record

**constructor** Create(aClient: TSQLRest; const aSQLWhere: RawUTF8); overload;

This constructor initializes the object as above, and fills its content from a client or server connection, using a specified WHERE clause
- the WHERE clause should use inlined parameters (like 'Name=('Arnaud'):') for better server speed - note that you can use FormatUTF8() as such:
  ```pascal
  aRec := TSQLRec.Create(Client,FormatUTF8('Salary>? AND Salary<?',[],[1000,2000]));
  or call the overloaded constructor with BoundsSQLWhere array of parameters
```

**constructor** Create(aClient: TSQLRest; const FormatSQLWhere: RawUTF8; const BoundsSQLWhere: array of const); overload;

This constructor initializes the object as above, and fills its content from a client or server connection, using a specified WHERE clause with parameters
- for better server speed, the WHERE clause should use bound parameters identified as '?' in the FormatSQLWhere statement, which is expected to follow the order of values supplied in BoundsSQLWhere open array - use DateToSQL/DateTimeToSQL for TDateTime, or directly any integer / double / currency / RawUTF8 values to be bound to the request as parameters
- note that this method prototype changed with revision 1.17 of the framework: array of const used to be ParamsSQLWhere and '%' in the FormatSQLWhere statement, whereas it now expects bound parameters as '?'

---

**mORMot.pas unit - Rev. 1.18**

**Page 1983 of 2552**
constructor CreateAndFillPrepare(aClient: TSQLRest; const aSQLWhere: RawUTF8; const aCustomFieldsCSV: RawUTF8=''); overload;

This constructor initializes the object as above, and prepares itself to loop through a statement using a specified WHERE clause
- this method creates a TSQLTableJSON, retrieves all records corresponding to the WHERE clause, then call FillPrepare - previous Create(aClient) methods retrieve only one record, this one more multiple rows
- you should then loop for all rows using 'while Rec.FillOne do ...'
- the TSQLTableJSON will be freed by TSQLRecord.Destroy
- the WHERE clause should use inlined parameters (like 'Name=:('Arnaud'):') for better server speed - note that you can use FormatUTF8() as such:
  aRec := TSQLMyRec.CreateAndFillPrepare(Client,FormatUTF8('Salary>? AND Salary<?',[],[1000,2000]));

or call the overloaded CreateAndFillPrepare() constructor directly with BoundsSQLWhere array of parameters
- aCustomFieldsCSV can be used to specify which fields must be retrieved
  - default aCustomFieldsCSV='' will retrieve all simple table fields
  - if aCustomFieldsCSV='*', it will retrieve all fields, including BLOBs
  - aCustomFieldsCSV can also be set to a CSV field list to retrieve only the needed fields, and save remote bandwidth - note that any later Update() will update all simple fields, so potentially with wrong values; but BatchUpdate() can be safely used since it will

constructor CreateAndFillPrepare(const aJSON: RawUTF8); overload;
This constructor initializes the object, and prepares itself to loop through a specified JSON table, which will use a private copy
- this method creates a TSQLTableJSON, fill it with the supplied JSON buffer, then call FillPrepare
- previous Create(aClient) methods retrieve only one record, this one more multiple rows
- you should then loop for all rows using 'while Rec.FillOne do ...'
- the TSQLTableJSON will be freed by TSQLRecord.Destroy

constructor CreateAndFillPrepare(aJSON: PUTF8Char; aJSONLen: integer); overload;
This constructor initializes the object, and prepares itself to loop through a specified JSON table buffer, which will be modified in-place
- this method creates a TSQLTableJSON, fill it with the supplied JSON buffer, then call FillPrepare
- previous Create(aClient) methods retrieve only one record, this one more multiple rows
- you should then loop for all rows using 'while Rec.FillOne do ...'
- the TSQLTableJSON will be freed by TSQLRecord.Destroy
**constructor** CreateAndFillPrepare(aClient: TSQLRest; const aIDs: array of Int64; const aCustomFieldsCSV: RawUTF8=''); overload;

This constructor initializes the object as above, and prepares itself to loop through a given list of IDs
- this method creates a TSQLTableJSON, retrieves all records corresponding to the specified IDs, then call FillPrepare - previous Create(aClient) methods retrieve only one record, this one more multiple rows
- you should then loop for all rows using 'while Rec.FillOne do ...'
- the TSQLTableJSON will be freed by TSQLRecord.Destroy
- aCustomFieldsCSV can be used to specify which fields must be retrieved
  - default aCustomFieldsCSV="" will retrieve all simple table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields
  - if aCustomFieldsCSV='*', it will retrieve all fields, including BLOBs
- note that you should not use this aCustomFieldsCSV optional parameter if you want to Update the retrieved record content later, since any missing fields will be left with previous values - but BatchUpdate() can be safely used after FillPrepare (will set only ID, TModTime and mapped fields)

**constructor** CreateAndFillPrepare(aClient: TSQLRest; const FormatSQLWhere: RawUTF8; const BoundsSQLWhere: array of const; const aCustomFieldsCSV: RawUTF8=''); overload;

This constructor initializes the object as above, and prepares itself to loop through a statement using a specified WHERE clause
- this method creates a TSQLTableJSON, retrieves all records corresponding to the WHERE clause, then call FillPrepare - previous Create(aClient) methods retrieve only one record, this one more multiple rows
- you should then loop for all rows using 'while Rec.FillOne do ...'
- the TSQLTableJSON will be freed by TSQLRecord.Destroy
- for better server speed, the WHERE clause should use bound parameters identified as '?' in the FormatSQLWhere statement, which is expected to follow the order of values supplied in BoundsSQLWhere open array - use DateToSQL/DateTimeToSQL for TDateTime, or directly any integer / double / currency / RawUTF8 values to be bound to the request as parameters
- note that this method prototype changed with revision 1.17 of the framework: array of const used to be ParamsSQLWhere and '%' in the FormatSQLWhere statement, whereas it now expects bound parameters as '?'
- aCustomFieldsCSV can be used to specify which fields must be retrieved
  - default aCustomFieldsCSV="" will retrieve all simple table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields
  - if aCustomFieldsCSV='*', it will retrieve all fields, including BLOBs
- note that you should not use this aCustomFieldsCSV optional parameter if you want to Update the retrieved record content later, since any missing fields will be left with previous values - but BatchUpdate() can be safely used after FillPrepare (will set only ID, TModTime and mapped fields)
constructor CreateAndFillPrepare(aClient: TSQLRest; const FormatSQLWhere: RawUTF8; const ParamsSQLWhere, BoundsSQLWhere: array of const; const aCustomFieldsCSV: RawUTF8=''); overload;

This constructor initializes the object as above, and prepares itself to loop through a statement using a specified WHERE clause
- this method creates a TSQLTableJSON, retrieves all records corresponding to the WHERE clause, then call FillPrepare - previous Create(aClient) methods retrieve only one record, this one more multiple rows
- you should then loop for all rows using 'while Rec.FillOne do ...'
- the TSQLTableJSON will be freed by TSQLRecord.Destroy
- the FormatSQLWhere clause will replace all '%' chars with the supplied ParamsSQLWhere[] supplied values, and bind all '?' chars as parameters with BoundsSQLWhere[] values
- aCustomFieldsCSV can be used to specify which fields must be retrieved
- default aCustomFieldsCSV="" will retrieve all simple table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields
- if aCustomFieldsCSV='*', it will retrieve all fields, including BLOBs
- note that you should not use this aCustomFieldsCSV optional parameter if you want to Update the retrieved record content later, since any missing fields will be left with previous values - but BatchUpdate() can be safely used after FillPrepare (will set only ID, TModTime and mapped fields)

constructor CreateAndFillPrepare Joined(aClient: TSQLRest; const aFormatSQLJoin: RawUTF8; const aParamsSQLJoin, aBoundsSQLJoin: array of const);

This constructor initializes the object, and prepares itself to loop nested TSQLRecord properties, through a JOINed statement and a WHERE clause
- by default, CreateAndFillPrepare() will return only the one-to-one nested TSQLRecord published properties IDs trans-typed as pointer - this constructor allow to retrieve the nested values in one statement
- this method creates a TSQLTableJSON, fill it with the supplied JSON buffer, then call FillPrepare
- previous CreateJoined() method retrieve only one record, this one more multiple rows
- you should then loop for all rows using 'while Rec.FillOne do ...'
- use this constructor if you want all TSQLRecord published properties to be allocated, and loaded with the corresponding values
- Free/Destroy will release these instances
- warning: if you call Update() after it, only the main object will be updated, not the nested TSQLRecord properties
**constructor** CreateAndFillPrepareMany(aClient: TSQLRest; const aFormatSQLJoin: RawUTF8; const aParamsSQLJoin, aBoundsSQLJoin: array of const);

This constructor initializes the object including all TSQLRecordMany properties, and prepares itself to loop through a JOINed statement
- the created instance will have all its TSQLRecordMany Dest property allocated with proper instance (and not only pointer(DestID) e.g.), ready to be consumed during a while FillOne do...
  loop (those instances will be freed by TSQLRecord.FillClose or Destory) - and the Source property won't contain pointer(SourceID) but the main TSQLRecord instance
- the aFormatSQLJoin clause will define a WHERE clause for an automated JOINed statement, including TSQLRecordMany published properties (and their nested properties)
- a typical use could be the following:
  aProd := TSQLProduct.CreateAndFillPrepareMany(Database,
  ['mark','for boy','small','medium']);
  if aProd<>nil then
  try
    while aProd.FillOne do
      // here e.g. aProd.Categories.Dest are instantiated (and Categories.Source=aProd)
      // you may also use aProd.FillTable to fill a grid, e.g.
      // (do not forget to set aProd.FillTable.OwnerMustFree := false)
    finally
      aProd.Free; // will also free aProd.Categories/Sizes instances
  end;
  this will execute a JOINed SELECT statement similar to the following:
  select p.*, c.*, s.*
  from Product p, Category c, Categories cc, Size s, Sizes ss
  where c.id=cc.dest and cc.source=p.id and
  s.id=ss.dest and ss.source=p.id and
  p.Owner='mark' and c.Name='for boy' and (s.Name='small' or s.Name='medium')
  - you SHALL call explicitly the FillClose method before using any methods of nested TSQLRecordMany instances which may override the Dest instance content (e.g. ManySelect) to avoid any GPF
  - the aFormatSQLJoin clause will replace all '%' chars with the supplied aParamsSQLJoin[] supplied values, and bind all '?' chars as bound parameters with aBoundsSQLJoin[] values

**constructor** CreateFrom(const aDocVariant: variant); overload;

This constructor initializes the object as above, and fills its content from a supplied TDocVariant object document
- is a wrapper around Create + FillFrom() methods

**constructor** CreateFrom(P: PUTF8Char); overload;

This constructor initializes the object as above, and fills its content from a supplied JSON buffer
- is a wrapper around Create + FillFrom() methods
- use JSON data, as exported by GetJSONValues(), expanded or not
- the data inside P^ is modified (unescape and transformed in-place): don't call CreateFrom(pointer(JSONRecord)) but CreateFrom(JSONRecord) which makes a temporary copy of the JSONRecord text variable before parsing

**constructor** CreateFrom(const JSONRecord: RawUTF8); overload;

This constructor initializes the object as above, and fills its content from a supplied JSON content
- is a wrapper around Create + FillFrom() methods
- use JSON data, as exported by GetJSONValues(), expanded or not
- make an internal copy of the JSONTable RawUTF8 before calling FillFrom() below
constructor CreateJoined(aClient: TSQLRest; aID: TID);

This constructor initializes the object from its ID, including all nested TSQLRecord properties, through a JOINed statement
- by default, Create(aClient,aID) will return only the one-to-one nested TSQLRecord published properties IDs trans-typed as pointer - this constructor allow to retrieve the nested values in one statement
- use this constructor if you want all TSQLRecord published properties to be allocated, and loaded with the corresponding values
- Free/Destroy will release these instances
- warning: if you call Update() after it, only the main object will be updated, not the nested TSQLRecord properties

destructor Destroy; override;

Release the associated memory
- in particular, release all TSQLRecordMany instance created by the constructor of this TSQLRecord

class function AutoFree(var localVariable; Rest: TSQLRest; const FormatSQLWhere: RawUTF8; const BoundsSQLWhere: array of const; const aCustomFieldsCSV: RawUTF8=""): IAutoFree; overload;

FillPrepare and protect one TSQLRecord local variable instance
- is a wrapper around TAutoFree.Create(localVariable,CreateAndFillPrepare(Rest,...))
- be aware that it won't implement a full ARC memory model, but may be just used to avoid writing some try ... finally blocks on local variables
- use with caution, only on well defined local scope
- warning: under FPC, you should assign the result of this method to a local IAutoFree variable, or use a with TSQLRecord.AutoFree() do statement - see http://bugs.freepascal.org/view.php?id=26602

class function AutoFree(var localVariable; Rest: TSQLRest; const FormatSQLWhere: RawUTF8; const ParamsSQLWhere,BoundsSQLWhere: array of const; const aCustomFieldsCSV: RawUTF8=""): IAutoFree; overload;

FillPrepare and protect one TSQLRecord local variable instance
- is a wrapper around TAutoFree.Create(localVariable,CreateAndFillPrepare(Rest,...))
- be aware that it won't implement a full ARC memory model, but may be just used to avoid writing some try ... finally blocks on local variables
- use with caution, only on well defined local scope
- warning: under FPC, you should assign the result of this method to a local IAutoFree variable, or use a with TSQLRecord.AutoFree() do statement - see http://bugs.freepascal.org/view.php?id=26602

class function AutoFree(var localVariable; Rest: TSQLRest; ID: TID): IAutoFree;

overload;

Read and protect one TSQLRecord local variable instance
- is a wrapper around TAutoFree.Create(localVariable,Create(Rest,ID))
- be aware that it won't implement a full ARC memory model, but may be just used to avoid writing some try ... finally blocks on local variables
- use with caution, only on well defined local scope
- warning: under FPC, you should assign the result of this method to a local IAutoFree variable, or use a with TSQLRecord.AutoFree() do statement - see http://bugs.freepascal.org/view.php?id=26602
class function AutoFree(varClassPairs: array of pointer): IAutoFree; overload;

Protect several TSQLRecord local variable instances
- specified as localVariable/recordClass pairs
- is a wrapper around TAutoFree.Several(...) constructor
- be aware that it won't implement a full ARC memory model, but may be just used to avoid writing some try ... finally blocks on local variables
- use with caution, only on well defined local scope
- you may write for instance:
  var info: TSQLBlogInfo;
  article: TSQLArticle;
  comment: TSQLComment;
  begin
    TSQLRecord.AutoFree([ // avoid several try..finally
      @info,TSQLBlogInfo,
      @article,TSQLArticle,
      @comment,TSQLComment]);
    .... now you can use info, article or comment
  end; // will call info.Free article.Free and comment.Free

- warning: under FPC, you should assign the result of this method to a local IAutoFree variable, or use a with TSQLRecord.AutoFree() do statement - see http://bugs.freepascal.org/view.php?id=26602

class function AutoFree(var localVariable): IAutoFree; overload;

Protect one TSQLRecord local variable instance
- be aware that it won't implement a full ARC memory model, but may be just used to avoid writing some try ... finally blocks on local variables
- use with caution, only on well defined local scope
- you may write for instance:
  var info: TSQLBlogInfo;
  begin
    TSQLBlogInfo.AutoFree(info);
    .... now you can use info
  end;

- warning: under FPC, you should assign the result of this method to a local IAutoFree variable, or use a with TSQLRecord.AutoFree() do statement - see http://bugs.freepascal.org/view.php?id=26602

class function CaptionName(Action: PRawUTF8=nil; ForHint: boolean=false): string;
virtual;

Get the captions to be used for this class
- if Action is nil, return the caption of the table name
- if Action is not nil, return the caption of this Action (lowercase left-trimmed)
- return "string" type, i.e. WideString for Delphi 2009+
- internally call UnCamelCase() then System.LoadResStringTranslate() if available
- ForHint is set to TRUE when the record caption name is to be displayed inside the popup hint of a button (i.e. the name must be fully qualified, not the default short version)
- is not part of TSQLRecordProperties because has been declared as virtual

class function CaptionNameFromRTTI(Action: PShortString): string;

Get the captions to be used for this class
- just a wrapper calling CaptionName() virtual method, from a ShortString pointer

function ClassProp: PClassProp;

Return the RTTI property information for this record

Used for DI-2.1.3 (page 2546).
function CreateCopy: TSQLRecord; overload; virtual;

This method create a clone of the current record, with same ID and properties
- copy all COPIABLE_FIELDS, i.e. all fields excluding tftMany (because those fields don't contain any data, but a TSQLRecordMany instance which allow to access to the pivot table data)
- you can override this method to allow custom copy of the object, including (or not) published properties copy

function CreateCopy(const CustomFields: TSQLFieldBits): TSQLRecord; overload;

This method create a clone of the current record, with same ID and properties
- overloaded method to copy the specified properties

function DynArray(const DynArrayFieldName: RawUTF8): TDynArray; overload;

Initialize a TDynArray wrapper to map dynamic array property values
- if the field name is not existing or not a dynamic array, result.IsVoid will be TRUE

function DynArray(DynArrayFieldName: RawUTF8): TDynArray; overload;

Initialize a TDynArray wrapper to map dynamic array property values
- this overloaded version expect the dynamic array to have been defined with a not null index attribute, e.g.
  
  published
  property Ints: TIntegerDynArray index 1 read fInts write fInts;
  property Currency: TCurrencyDynArray index 2 read fCurrency write fCurrency;

- if the field index is not existing or not a dynamic array, result.IsVoid will be TRUE

function EnginePrepareMany(aClient: TSQLRest; const aFormatSQLJoin: RawUTF8; const aParamsSQLJoin, aBoundsSQLJoin: array of const; out ObjectsClass: TSQLRecordClassDynArray; out SQL: RawUTF8): RawUTF8;

Compute a JOINed statement including TSQLRecordMany fields
- is called by FillPrepareMany() to retrieve the JSON of the corresponding request: so you could use this method to retrieve directly the same information, ready to be transmitted (e.g. as RawJSON) to a client

function FillOne(aDest: TSQLRecord=nil): boolean;

Fill all published properties of this object from the next available TSQLTable prepared row
- FillPrepare() must have been called before
- the Row number is taken from property FillCurrentRow
- return true on success, false if no more Row data is available
- internally call FillRow() to update published properties values
function FillPrepare(aClient: TSQLRest; const aSQLWhere: RawUTF8=''; const aCustomFieldsCSV: RawUTF8=''; aCheckTableName: TSQLCheckTableName=ctnNoCheck): boolean; overload;

Prepare to get values from a SQL where statement
- returns true in case of success, false in case of an error during SQL request
- then call FillRow(1..Table.RowCount) to get any row value
- or you can also loop through all rows with
  while Rec.FillOne do
do something with(Rec);
- a temporary TSQLTable is created then stored in an internal fTable protected field
- if aSQLWhere is left to '', all rows are retrieved as fast as possible (e.g. by-passing SQLite3 virtual table modules for external databases)
- the WHERE clause should use inlined parameters (like 'Name:=('Arnaud'):') for better server speed - note that you can use FormatUTF8() as such:
  aRec.FillPrepare(Client,FormatUTF8('Salary>? AND Salary<?',[],[1000,2000]));
- aCustomFieldsCSV can be used to specify which fields must be retrieved
  - default aCustomFieldsCSV='' will retrieve all simple table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields
  - if aCustomFieldsCSV='*', it will retrieve all fields, including BLOBs
  - note that you should not use this aCustomFieldsCSV optional parameter if you want to Update the retrieved record content later, since any missing fields will be left with previous values - but BatchUpdate() can be safely used after FillPrepare (will set only ID, TModTime and mapped fields)

or call the overloaded FillPrepare() method directly with BoundsSQLWhere array of parameters
- aCustomFieldsCSV can be used to specify which fields must be retrieved
- default aCustomFieldsCSV='' will retrieve all simple table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields
- if aCustomFieldsCSV='*', it will retrieve all fields, including BLOBs
- note that you should not use this aCustomFieldsCSV optional parameter if you want to Update the retrieved record content later, since any missing fields will be left with previous values - but BatchUpdate() can be safely used after FillPrepare (will set only ID, TModTime and mapped fields)

function FillPrepare(aClient: TSQLRest; const FormatSQLWhere: RawUTF8; const BoundsSQLWhere: array of const; const aCustomFieldsCSV: RawUTF8=''): boolean; overload;

Prepare to get values using a specified WHERE clause with '%' parameters
- returns true in case of success, false in case of an error during SQL request
- then call FillRow(1..Table.RowCount) to get any row value
- or you can also loop through all rows with
  while Rec.FillOne do
do something with(Rec);
- a temporary TSQLTable is created then stored in an internal fTable protected field
- for better server speed, the WHERE clause should use bound parameters identified as '?' in the FormatSQLWhere statement, which is expected to follow the order of values supplied in BoundsSQLWhere open array - use DateToSQL/DateTimeToSQL for TDateTime, or directly any integer / double / currency / RawUTF8 values to be bound to the request as parameters
- note that this method prototype changed with revision 1.17 of the framework: array of const used to be ParamsSQLWhere and '%' in the FormatSQLWhere statement, whereas it now expects bound parameters as '?'
- aCustomFieldsCSV can be used to specify which fields must be retrieved
  - default aCustomFieldsCSV='' will retrieve all simple table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields
  - if aCustomFieldsCSV='*', it will retrieve all fields, including BLOBs
  - note that you should not use this aCustomFieldsCSV optional parameter if you want to Update the retrieved record content later, since any missing fields will be left with previous values - but BatchUpdate() can be safely used after FillPrepare (will set only ID, TModTime and mapped fields)
function FillPrepare(aClient: TSQLRest; const FormatSQLWhere: RawUTF8; const ParamsSQLWhere, BoundsSQLWhere: array of const; const aCustomFieldsCSV: RawUTF8=''): boolean; overload;

Prepare to get values using a specified WHERE clause with '%' and '?' parameters
- returns true in case of success, false in case of an error during SQL request
- then call FillRow(1..Table.RowCount) to get any row value
- or you can also loop through all rows with
  while Rec.FillOne do
    dosomethingwith(Rec);

- a temporary TSQLTable is created then stored in an internal fTable protected field
- the FormatSQLWhere clause will replace all '%' chars with the supplied ParamsSQLWhere[] supplied values, and bind all '?' chars as bound parameters with BoundsSQLWhere[] values
- aCustomFieldsCSV can be used to specify which fields must be retrieved
- default aCustomFieldsCSV="" will retrieve all simple table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields
- if aCustomFieldsCSV="*", it will retrieve all fields, including BLOBs
- note that you should not use this aCustomFieldsCSV optional parameter if you want to Update the retrieved record content later, since any missing fields will be left with previous values - but BatchUpdate() can be safely used after FillPrepare (will set only ID, TModTime and mapped fields)

function FillPrepare(aClient: TSQLRest; const aIDs: array of Int64; const aCustomFieldsCSV: RawUTF8=''): boolean; overload;

Prepare to get values from a list of IDs
- returns true in case of success, false in case of an error during SQL request
- then call FillRow(1..Table.RowCount) to get any row value
- or you can also loop through all rows with
  while Rec.FillOne do
    dosomethingwith(Rec);

- a temporary TSQLTable is created then stored in an internal fTable protected field
- aCustomFieldsCSV can be used to specify which fields must be retrieved
- default aCustomFieldsCSV="" will retrieve all simple table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields
- if aCustomFieldsCSV="*", it will retrieve all fields, including BLOBs
- note that you should not use this aCustomFieldsCSV optional parameter if you want to Update the retrieved record content later, since any missing fields will be left with previous values - but BatchUpdate() can be safely used after FillPrepare (will set only ID, TModTime and mapped fields)
function FillPrepareMany(aClient: TSQLRest; const aFormatSQLJoin: RawUTF8; const aParamsSQLJoin, aBoundsSQLJoin: array of const): boolean;

// prepare to loop through a JOINed statement including TSQLRecordMany fields
- all TSQLRecordManyDest published fields will now contain a true TSQLRecord instance, ready to be filled with the JOINed statement results (these instances will be released at FillClose) - the same for Source which will point to the self instance
- the aFormatSQLJoin clause will define a WHERE clause for an automated JOINed statement, including TSQLRecordMany published properties (and their nested properties)
- returns true in case of success, false in case of an error during SQL request
- a typical use could be the following:
  if aProd.FillPrepareMany(Database,'Owner=? and Categories.Dest.Name=? and (Sizes.Dest.Name=? or Sizes.Dest.Name=?)',[],['mark','','small','medium']) then
    while aProd.FillOne do
      // here e.g. aProd.Categories.Dest are instantiated (and Categories.Source=aProd)
      writeln(aProd.Name,' ',aProd.Owner,' ',aProd.Categories.Dest.Name,'
      // you may also use aProd.FillTable to fill a grid, e.g.
      // (do not forget to set aProd.FillTable.OwnerMustFree := false)
  this will execute a JOINed SELECT statement similar to the following:
  select p.*, c.*, s.*
  from Product p, Category c, Categories cc, Size s, Sizes ss
  where c.id=cc.dest and cc.source=p.id and
  s.id=ss.dest and ss.source=p.id and
  p.Owner='mark' and c.Name='for boy' and (s.Name='small' or s.Name='medium')
  - the FormatSQLWhere clause will replace all '%' chars with the supplied ParamsSQLWhere[] supplied values, and bind all '?' chars as parameters with BoundsSQLWhere[] values
  - you SHALL call explicitly the FillClose method before using any methods of nested TSQLRecordMany instances which may override the Dest instance content (e.g. ManySelect) to avoid any GPF
  - is used by TSQLRecord.CreateAndFillPrepareMany constructor

function FillRewind: boolean;

Go to the first prepared row, ready to loop through all rows with FillOne()
- the Row number (property FillCurrentRow) is reset to 1
- return true on success, false if no Row data is available
- you can use it e.g. as:
  while Rec.FillOne do
    dosomethingwith(Rec);
  if Rec.FillRewind then
    while Rec.FillOne do
      dosomeotherthingwith(Rec);
function FillRow(aRow: integer; aDest: TSQLRecord=nil): boolean; virtual;

 Fill all published properties of an object from a TSQLTable prepared row
- FillPrepare() must have been called before
- if Dest is nil, this object values are filled
- if Dest is not nil, this object values will be filled, but it won't work with TSQLRecordMany
  properties (i.e. after FillPrepareMany call)
- ID field is updated if first Field Name is 'ID'
- Row number is from 1 to Table.RowCount
- setter method (write Set*) is called if available
- handle UTF-8 SQL to Delphi values conversion (see TPropInfo mapping)
- this method has been made virtual e.g. so that a calculated value can be used in a custom field

function Filter(const aFields: TSQLFieldBits=[0..MAX_SQLFIELDSD-1]): boolean; overload; virtual;

 Filter/transform the specified fields values of the TSQLRecord instance
- by default, this will perform all TSynFilter as registered by [RecordProps.]AddFilterOrValidate()
- inherited classes may add some custom filtering/transformation here, if it's not needed nor
  mandatory to create a new TSynFilter class type: in this case, the function has to return TRUE if
  the filtering took place, and FALSE if any default registered TSynFilter must be processed
- the default aFields parameter will process all fields

function Filter(const aFields: array of RawUTF8): boolean; overload;

 Filter/transform the specified fields values of the TSQLRecord instance
- this version will call the overloaded Filter() method above
- return TRUE if all field names were correct and processed, FALSE otherwise

function FilterAndValidate(aRest: TSQLRest; const aFields: TSQLFieldBits=[0..MAX_SQLFIELDSD-1]; aValidator: PSynValidate=nil): RawUTF8; overload;

 Filter (transform) then validate the specified fields values of the TSQLRecord
- this version will call the overloaded Filter() and Validate() methods and return '' on validation
  success, or an error message with the faulty field names at the beginning

function FilterAndValidate(aRest: TSQLRest; out aErrorMessage: string; const aFields: TSQLFieldBits=[0..MAX_SQLFIELDSD-1]; aValidator: PSynValidate=nil): boolean; overload;

 Filter (transform) then validate the specified fields values of the TSQLRecord
- this version will call the overloaded Filter() and Validate() methods and display the faulty field
  name at the beginning of the error message
- returns true if all field names were correct and processed, or false and an explicit error
  message (translated in the current language) on error

function GetAsDocVariant(withID: boolean; const withFields: TSQLFieldBits; options: PDocVariantOptions=nil; replaceRowIDWithID: boolean=false): variant; overload;

 Retrieve the record content as a TDocVariant custom variant object

function GetBinary: RawByteString;

 Write the record fields into RawByteString a binary buffer
- same as GetBinaryValues(), but also writing the ID field first
function GetFieldValue(const PropName: RawUTF8): RawUTF8;

Retrieve a field value from a given property name, as encoded UTF-8 text
- you should use strong typing and direct property access, following the ORM approach of the framework; but in some cases (a custom Grid display, for instance), it could be useful to have this method available
- will return '' in case of wrong property name
- BLOB and dynamic array fields are returned as '\uFFF0base64encodedbinary'

function GetFieldVariant(const PropName: string): Variant;

Retrieve the published property value into a Variant
- will set the Variant type to the best matching kind according to the property type
- will return a null variant in case of wrong property name
- BLOB fields are returned as SQLite3 BLOB literals ("x'01234'" e.g.)
- dynamic array fields are returned as a Variant array

function GetJSONValues(Expand, withID: boolean; Occasion: TSQLOccasion; UsingStream: TCustomMemoryStream=nil; SQLRecordOptions: TJSONSerializerSQLRecordOptions=[]): RawUTF8; overload;

Same as overloaded GetJSONValues(), but returning result into a RawUTF8
- if UsingStream is not set, it will use a temporary THeapMemoryStream instance

function GetJSONValues(Expand, withID: boolean; const Fields: TSQLFieldBits; SQLRecordOptions: TJSONSerializerSQLRecordOptions=[]): RawUTF8; overload;

Same as overloaded GetJSONValues(), but allowing to set the fields to be retrieved, and returning result into a RawUTF8

function GetJSONValues(Expand, withID: boolean; const FieldsCSV: RawUTF8; SQLRecordOptions: TJSONSerializerSQLRecordOptions=[]): RawUTF8; overload;

Same as overloaded GetJSONValues(), but allowing to set the fields to be retrieved, and returning result into a RawUTF8

function GetNonVoidFields: TSQLFieldBits;

Set the bits corresponding to non-void (0,"" ) copiable fields

function GetSimpleFieldsAsDocVariant(withID: boolean=true; options: PDocVariantOptions=nil): variant;

Retrieve the simple record content as a TDocVariant custom variant object
class function GetSQLCreate(aModel: TSQLModel): RawUTF8; virtual;
    Return the UTF-8 encoded SQL source to create the table containing the published fields of a TSQLRecord child
    - a 'ID INTEGER PRIMARY KEY' field is always created first (mapping SQLite3 RowID)
    - AnsiString are created as TEXT COLLATE NOCASE (fast SQLite3 7bits compare)
    - RawUnicode and RawUTF8 are created as TEXT COLLATE SYSTEMNOCASE (i.e. use our fast UTF8IComp() for comparison)
    - TDateTime are created as TEXT COLLATE ISO8601 (which calls our very fast ISO TEXT to Int64 conversion routine)
    - an individual bit set in UniqueField forces the corresponding field to be marked as UNIQUE (an unique index is automatically created on the specified column); use TSQLModel fIsUnique[] array, which set the bits values to 1 if a property field was published with "stored AS UNIQUE" (i.e. "stored false")
    - this method will handle TSQLRecordFTS* classes like FTS* virtual tables, TSQLRecordRTree as RTREE virtual table, and TSQLRecordVirtualTable*ID classes as corresponding Delphi designed virtual tables
    - is not part of TSQLRecordProperties because has been declared as virtual so that you could specify a custom SQL statement, per TSQLRecord type
    - anyway, don't call this method directly, but use TSQLModel.GetSQLCreate()
    - the aModel parameter is used to retrieve the Virtual Table module name, and can be ignored for regular (not virtual) tables

function GetSQLSet: RawUTF8;
    Return the UTF-8 encoded SQL source to UPDATE the values contained in the current published fields of a TSQLRecord child
    - only simple fields name (i.e. not TSQLRawBlob/TSQLRecordMany) are retrieved: BLOB fields are ignored (use direct access via dedicated methods instead)
    - format is 'COL1='VAL1', COL2='VAL2''
    - is not used by the ORM (do not use prepared statements) - only here for conveniency

function GetSQLValues: RawUTF8;
    Return the UTF-8 encoded SQL source to INSERT the values contained in the current published fields of a TSQLRecord child
    - only simple fields name (i.e. not TSQLRawBlob/TSQLRecordMany) are updated: BLOB fields are ignored (use direct update via dedicated methods instead)
    - format is '(COL1, COL2) VALUES ('VAL1', 'VAL2')' if some column was ignored (BLOB e.g.)
    - format is 'VALUES ('VAL1', 'VAL2')' if all columns values are available
    - is not used by the ORM (do not use prepared statements) - only here for conveniency

function RecordClass: TSQLRecordClass;
    Return the Class Type of the current TSQLRecord

class function RecordProps: TSQLRecordProperties;
    Direct access to the TSQLRecord properties from RTTI
    - TSQLRecordProperties is faster than e.g. the class function FieldProp()
    - use internal the unused vmtAutoTable VMT entry to fast retrieve of a class variable which is unique for each class ("class var" is unique only for the class within it is defined, and we need a var for each class: so even Delphi XE syntax is not powerful enough for our purpose, and the vmtAutoTable trick if very fast, and works with all versions of Delphi - including 64-bit target)
function RecordReference(Model: TSQLModel): TRecordReference;
Return the TRecordReference Int64 value pointing to this record

function SameRecord(Reference: TSQLRecord): boolean;
Return true if all published properties values in Other are identical to the published properties of this object
- instances must be of the same class type
- only simple fields (i.e. not TSQLRawBlob/TSQLRecordMany) are compared
- comparison is much faster than SameValues() below

function SameValues(Reference: TSQLRecord): boolean;
Return true if all published properties values in Other are identical to the published properties of this object
- work with different classes: Reference properties name must just be present in the calling object
- only simple fields (i.e. not TSQLRawBlob/TSQLRecordMany) are compared
- compare the text representation of the values: fields may be of different type, encoding or precision, but still have same values

function SetBinary(P, PEnd: PAnsiChar): Boolean; overload;
Set the record fields from a binary buffer saved by GetBinary()
- same as SetBinaryValues(), but also reading the ID field first
- PEnd should point to the end of the P input buffer, to avoid any overflow

function SetBinary(const binary: RawByteString): Boolean; overload;
Set the record fields from a binary buffer saved by GetBinary()
- same as SetBinaryValues(), but also reading the ID field first

function SetBinaryValues(var P: PAnsiChar; PEnd: PAnsiChar): Boolean;
Set the field values from a binary buffer
- won't read the ID field (should be read before, with the Count e.g.)
- PEnd should point just after the P input buffer, to avoid buffer overflow
- returns true on success, or false in case of invalid content in P^ e.g.
- P is updated to the next pending content after the read values

function SetBinaryValuesSimpleFields(var P: PAnsiChar; PEnd: PAnsiChar): Boolean;
Set the simple field values from a binary buffer
- won't read the ID field (should be read before, with the Count e.g.)
- PEnd should point just after the P input buffer, to avoid buffer overflow
- returns true on success, or false in case of invalid content in P^ e.g.
- P is updated to the next pending content after the read values,

function SetFieldSQLVars(const Values: TSQLVarDynArray): boolean;
Set all field values from a supplied array of TSQLVar values
- Values[] array must match the RecordProps.Field[] order: will return false if the Values[].VType does not match RecordProps.FieldType[]
function SimplePropertiesFill(const aSimpleFields: array of const): boolean;
Set the simple fields with the supplied values
- the aSimpleFields parameters must follow explicitly the order of published properties of the
  supplied aTable class, excepting the TSQLRawBlob and TSQLRecordMany kind (i.e. only so called
  "simple fields") - in particular, parent properties must appear first in the list
- the aSimpleFields must have exactly the same count of parameters as there are "simple fields"
  in the published properties
- return true on success, but be aware that the field list must match the field layout, otherwise if
  may return true but will corrupt data

class function SQLTableName: RawUTF8;
The Table name in the database, associated with this TSQLRecord class
- 'TSQL' or 'TSQLRecord' chars are trimmed at the beginning of the ClassName
- or the ClassName is returned as is, if no 'TSQL' or 'TSQLRecord' at first
- is just a wrapper around RecordProps.SQLTableName

function Validate(aRest: TSQLRest; const aFields: array of RawUTF8;
aInvalidFieldIndex: PInteger=nil; aValidator: PSynValidate=nil): string; overload;
Validate the specified fields values of the current TSQLRecord instance
- this version will call the overloaded Validate() method above
- returns " if all field names were correct and processed, or an explicit error message (translated
  in the current language) on error
- if aInvalidFieldIndex is set, it will contain the first invalid field index

function Validate(aRest: TSQLRest; const aFields:
TSQLFieldBits=[0..MAX_SQLFIELDS-1]; aInvalidFieldIndex: PInteger=nil; aValidator:
PSynValidate=nil): string; overload; virtual;
Validate the specified fields values of the current TSQLRecord instance
- by default, this will perform all TSynValidate as registered by
  [RecordProps.]AddFilterOrValidate()
- it will also check if any UNIQUE field value won't be duplicated
- inherited classes may add some custom validation here, if it's not needed nor mandatory to
  create a new TSynValidate class type: in this case, the function has to return an explicit error
  message (as a generic VCL string) if the custom validation failed, or " if the validation was
  successful: in this later case, all default registered TSynValidate are processed
- the default aFields parameter will process all fields
- if aInvalidFieldIndex is set, it will contain the first invalid field index found
- caller SHOULD always call the Filter() method before calling Validate()

class procedure AddFilterNotVoidAllTextFields;
Register a TSynFilterTrim and a TSynValidateText filters so that all text fields, after space
trimming, won't be void
- will only affect RAWTEXT_FIELDS

class procedure AddFilterNotVoidText(const aFieldNames: array of RawUTF8);
Register a TSynFilterTrim and a TSynValidateText filters so that the specified fields, after space
trimming, won't be void
class procedure AddFilterOrValidate(const aFieldName: RawUTF8; aFilter: TSynFilterOrValidate);

Register a custom filter (transformation) or validate to the TSQLRecord class for a specified field
- this will be used by TSQLRecord.Filter and TSQLRecord.Validate methods (in default implementation)
- will raise an EModelException on failure
- this function is just a wrapper around RecordProps.AddFilterOrValidate

procedure AppendAsJsonObject(W: TJSONSerializer; Fields: TSQLFieldBits=[]);

Will append the record fields as an expanded JSON object
- GetJsonValues() will expect a dedicated TJSONSerializer, whereas this method will add the JSON object directly to any TJSONSerializer
- by default, will append the simple fields, unless the Fields optional parameter is customized to a non void value

procedure AppendFillAsJsonArray(const FieldName: RawUTF8; W: TJSONSerializer; Fields: TSQLFieldBits=[]);

Will append all the FillPrepare() records as an expanded JSON array
- generates '[[rec1],[rec2],...]' using a loop similar to:
  while FillOne do .. AppendAsJsonObject() ..
- if FieldName is set, the JSON array will be written as a JSON property, i.e. surrounded as "FieldName":[...,]' - note the ',' at the end
- by default, will append the simple fields, unless the Fields optional parameter is customized to a non void value
- see also TSQLRest.AppendListAsJsonArray for a high-level wrapper method

procedure AppendFillAsJsonValues(W: TJSONSerializer);

Will iterate over all FillPrepare items, appending them as a JSON array
- creates a JSON array of all record rows, using
  while FillOne do GetJSONValues(W)...

procedure ClearProperties(const aFieldsCSV: RawUTF8); overload;

Clear the values of specified published properties
- " will leave the content untouched, '*' will clear all simple fields

procedure ClearProperties; overload;

Clear the values of all published properties, and also the ID property

procedure ComputeFieldsBeforeWrite(aRest: TSQLRest; aOccasion: TSQLEvent);
virtual;
Should modify the record content before writing to the Server
- this default implementation will update any sftModTime / TModTime, sftCreateTime / TCreateTime and sftSessionUserID / TSessionUserID properties content with the exact server time stamp
- you may override this method e.g. for custom calculated fields
- note that this is computed only on the Client side, before sending back the content to the remote Server: therefore, TModTime / TCreateTime fields are a pure client ORM feature - it won't work directly at REST level
procedure FillClose;
Close any previous FillPrepare..FillOne loop
- is called implicitly by FillPrepare() call to release any previous loop
- release the internal hidden TSQLTable instance if necessary
- is not mandatory if the TSQLRecord is released just after, since TSQLRecord.Destroy will call it
- used e.g. by FillFrom methods below to avoid any GPF/memory confusion

procedure FillFrom(Table: TSQLTable; Row: integer); overload;
Fill all published properties of this object from a TSQLTable result row
- call FillPrepare() then FillRow(Row)

procedure FillFrom(const JSONRecord: RawUTF8; FieldBits: PSQLFieldBits=nil); overload;
Fill all published properties of this object from a JSON object result
- use JSON data, as exported by GetJSONValues()
- JSON data may be expanded or not
- make an internal copy of the JSONTable RawUTF8 before calling FillFrom() below
- if FieldBits is defined, it will store the identified field index

procedure FillFrom(const JSONTable: RawUTF8; Row: integer); overload;
Fill all published properties of this object from a JSON result row
- create a TSQLTable from the JSON data
- call FillPrepare() then FillRow(Row)

procedure FillFrom(aRecord: TSQLRecord; const aRecordFieldBits: TSQLFieldBits); overload;
Fill the specified properties of this object from another object
- source object must be a parent or of the same class as the current record
- copy the fields, as specified by their bit index in the source record; you may use aRecord.GetNonVoidFields if you want to update some fields

procedure FillFrom(aRecord: TSQLRecord); overload;
Fill all published properties of this object from another object
- source object must be a parent or of the same class as the current record
- copy all COPIABLE_FIELDS, i.e. all fields excluding tftMany (because those fields don't contain any data, but a TSQLRecordMany instance which allow to access to the pivot table data)

procedure FillFrom(const aDocVariant: variant); overload;
Fill all published properties of this object from a supplied TDocVariant object document
- is a wrapper around VariantSaveJSON() + FillFrom() methods

procedure FillFrom(P: PUTF8Char; FieldBits: PSQLFieldBits=nil); overload;
Fill all published properties of this object from a JSON result
- the data inside P^ is modified (unescape and transformed); don't call FillFrom(pointer(JSONRecordUTF8)) but FillFrom(JSONRecordUTF8) which makes a temporary copy of the JSONRecordUTF8 text
- use JSON data, as exported by GetJSONValues()
- JSON data may be expanded or not
- if FieldBits is defined, it will store the identified field index
procedure FillPrepare(Table: TSQLTable; aCheckTableName: TSQLCheckTableName=ctnNoCheck); overload;

Prepare to get values from a TSQLTable result
- then call FillRow(1.Table.RowCount) to get any row value
- or you can also loop through all rows with
  while Rec.FillOne do
dosomethingwith(Rec);
- the specified TSQLTable is stored in an internal fTable protected field
- set aCheckTableName if you want e.g. the Field Names from the Table any pending 'TableName.' trimmed before matching to the current record

procedure FillValue(PropName, Value: PUTF8Char; wasString: boolean; FieldBits: PSQLFieldBits=nil);

Fill a published property value of this object from a UTF-8 encoded value
- see TPropInfo about proper Delphi / UTF-8 type mapping/conversion
- use this method to fill a BLOB property, i.e. a property defined with type TSQLRawBlob, since by default all BLOB properties are not set by the standard Retrieve() method (to save bandwidth)
- if FieldBits is defined, it will store the identified field index

procedure ForceVariantFieldsOptions(aOptions: TDocVariantOptions=JSON_OPTIONS_FAST);

Change TDocVariantData.Options for all variant published fields
- may be used to replace e.g. JSON_OPTIONS_FAST_EXTENDED by JSON_OPTIONS_FAST

procedure GetAsDocVariant(withID: boolean; const withFields: TSQLFieldBits; var result: variant; options: PDocVariantOptions=nil; ReplaceRowIDWithID: boolean=false); overload;

Retrieve the record content as a TDocVariant custom variant object

procedure GetBinaryValues(W: TFileBufferWriter); overload;

Write the field values into the binary buffer
- won't write the ID field (should be stored before, with the Count e.g.)

procedure GetBinaryValues(W: TFileBufferWriter; const aFields: TSQLFieldBits); overload;

Write the field values into the binary buffer
- won't write the ID field (should be stored before, with the Count e.g.)

procedure GetBinaryValuesSimpleFields(W: TFileBufferWriter);

Write the simple field values (excluding ID) into the binary buffer

procedure GetJSONValues(W : TJSONSerializer); overload;

Return the UTF-8 encoded JSON objects for the values of this TSQLRecord
- layout and fields should have been set at TJSONSerializer construction: to append some content to an existing TJsonSerializer, call the AppendAsJsonObject() method

Used for DI-2.1.2 (page 2545), DI-2.1.3 (page 2546).
procedure GetJSONValues(JSON: TStream; Expand, withID: boolean; Occasion: TSQLOccasion; SQLRecordOptions: TJSONSerializerSQLRecordOptions=[]); overload;

Return the UTF-8 encoded JSON objects for the values contained in the current published fields of a TSQLRecord child
- only simple fields (i.e. not TSQLRawBlob/TSQLRecordMany) are retrieved: BLOB fields are ignored (use direct access via dedicated methods instead)
- if Expand is true, JSON data is an object, for direct use with any Ajax or .NET client:
  \{
    "col1": val11, "col2": "val12"
  \}
- if Expand is false, JSON data is serialized (as used in TSQLTableJSON)
  \{
    "fieldCount":1, "values": ["col1", "col2", val11, "val12", val21, ... ]
  \}
- if withID is true, then the first ID field value is included
- you can customize SQLRecordOptions, e.g. if sftObject/sftBlobDynArray property instance will be serialized as a JSON object or array, not a JSON string (which is the default, as expected by the database storage), or if an "ID_str" string field should be added for JavaScript

Used for DI-2.1.2 (page 2545), DI-2.1.3 (page 2546).

procedure GetJSONValuesAndFree(JSON : TJSONSerializer); overload;

Return the UTF-8 encoded JSON objects for the values of this TSQLRecord
- the JSON buffer will be finalized if needed (e.g. non expanded mode), and the supplied TJSONSerializer instance will be freed by this method
- layout and fields should have been set at TJSONSerializer construction: to append some content to an existing TJSONSerializer, call the AppendAsJsonObject() method

class procedure InitializeTable(Server: TSQLRestServer; const FieldName: RawUTF8; Options: TSQLInitializeTableOptions); virtual;

Virtual method called when the associated table is created in the database
- if FieldName is "", initialization regarding all fields must be made; if FieldName is specified, initialization regarding this field must be processed
- override this method in order to initialize indexes or create default records
- by default, create indexes for all TRecordReference properties, and for all TSQLRecord inherited properties (i.e. of sftID type, that is an INTEGER field containing the ID of the pointing record)
- the options specified at CreateMissingTables() are passed to this method, within the context of an opened DB transaction, in which missing tables and fields have already been added
- is not part of TSQLRecordProperties because has been declared as virtual

procedure SetFieldValue(const PropName: RawUTF8; Value: PUTF8Char);

Set a field value of a given property name, from some encoded UTF-8 text
- you should use strong typing and direct property access, following the ORM approach of the framework; but in some cases (a custom Grid display, for instance), it could be useful to have this method available
- won't do anything in case of wrong property name
- expect BLOB and dynamic array fields encoded as SQliite3 BLOB literals ("x'01234'" e.g.) or '\uFF00base64encodedbinary'

procedure SetFieldVariant(const PropName: string; const Source: Variant);

Set the published property value from a Variant value
- will convert from the variant type into UTF-8 text before setting the value (so will work with any kind of Variant)
- won't do anything in case of wrong property name
- expect BLOB fields encoded as SQLite3 BLOB literals ('\x01234' e.g.)
property AsTSQLRecord: pointer read GetIDAsPointer;

This read-only property can be used to retrieve the ID as a TSQLRecord object
- published properties of type TSQLRecord (one-to-many relationship) do not store real class instances (only exception is if they inherit from TSQLRecordMany) - you can use this value to assign a TSQLRecord instance to a published property, as such:

```
Main := TSQLRecordMain.Create;
Client.Add(Main);
Detail := TSQLRecordDetail.Create;
Detail.Main := Main.AsTSQLRecord;  // will store Main.ID in MAIN column
Client.Add(Detail);
```

- is especially useful on 64-bit platform, since on 32-bit:

```
Detail.Main := pointer(Main.ID)
```

compiles (whereas it won't on 64-bit) and is the same than platform-independent

```
Detail.Main := Main.AsTSQLRecord;
```

- using Main.AsTSQLRecord will ensure that the ID is retrieved, even if Main itself is not a true instance
- if the stored ID is bigger than 32-bit, then it will raise an EORMException: in this case, you should use a TID / T*ID kind of published property, and not a TSQLRecord, which is limited to the pointer size

property FillContext: TSQLRecordFill read fFill;

Used internally by FillPrepare() and corresponding Fill*() methods

property FillCurrentRow: integer read GetFillCurrentRow;

This property contains the current row number (beginning with 1), initialized to 1 by FillPrepare(), which will be read by FillOne

property FillReachedEnd: boolean read GetFillReachedEnd;

This property is set to true, if all rows have been browsed after FillPrepare / while FillOne do ...

property FillTable: TSQLTable read GetTable;

This property contains the TSQLTable after a call to FillPrepare()

property HasBlob: boolean read GetHasBlob;

This property is set to true, if any published property is a BLOB (TSQLRawBlob)

property ID: TID read GetID;

This property stores the record's integer ID
- if this TSQLRecord is not a instance, but a field value in a published property of type sftID (i.e. TSQLRecord(aID)), this method will try to retrieve it; but preferred method is to typecast it via PTrInt(aProperty), because GetID() relies on some low-level Windows memory mapping trick, and will recognize an ID value up to 1,048,576 (i.e. $100000)
- notice: the Setter should not be used usualy; you should not have to write aRecord.ID := someID in your code, since the ID is set during Retrieve or Add of the record
- use IDValue property for direct read/write access to the record's ID field, if you know that this TSQLRecord is a true allocated class instance

property IDValue: TID read fID write fID;

This property gives direct access to the record's integer ID
- using IDValue expects this TSQLRecord to be a true instance, not a transtyped sftID (i.e. TSQLRecord(aID))
property InternalState: cardinal read fInternalState;

This property contains the internal state counter of the server database when the data was retrieved from it.
- can be used to check if retrieved data may be out of date

property SimpleFieldCount: integer read GetSimpleFieldCount;

This property returns the published property count with any valid database field except TSQLRawBlob/TSQLRecordMany.
- by default, the TSQLRawBlob (BLOB) fields are not included into this set: they must be read specifically (in order to spare bandwidth).
- TSQLRecordMany fields are not accessible directly, but as instances created by TSQLRecord.Create.

TSQLRecordNoCase = class(TSQLRecord)

Root class for defining and mapping database records with case-insensitive NOCASE collation.
- abstract ancestor, from which you may inherit your own ORM classes.
- by default, any sftUTF8Text field (RawUTF8, UnicodeString, WideString properties) will use our Unicode SYSTEMNOCASE SQLite3 collation, which calls UTF8ILComp() to handle most western languages, but is not standard.
- you may inherit from this class to ensure any text field will use the faster and SQLite3 built-in NOCASE collation, handling only 7-bit A-Z chars.
- inherit from TSQLRecordNoCase or TSQLRecordCaseSensitive if you expect your text fields to contain only basic (un)accentued ASCCI characters, and to be opened by any standard/SQLite3 library or tool (outside of SynSQLite3.pas/SynDBExplorer).

TSQLRecordCaseSensitive = class(TSQLRecord)

Root class for defining and mapping database records with case-sensitive BINARY collation.
- abstract ancestor, from which you may inherit your own ORM classes.
- by default, any sftUTF8Text field (RawUTF8, UnicodeString, WideString properties) will use our Unicode SYSTEMNOCASE SQLite3 collation, which calls UTF8ILComp() to handle most western languages, but is not standard.
- you may inherit from this class to ensure any text field will use the faster and SQLite3 built-in BINARY collation, which is case-sensitive.
- inherit from TSQLRecordNoCase or TSQLRecordCaseSensitive if you expect your text fields to contain only basic (un)accentued ASCCI characters, and to be opened by any standard/SQLite3 library or tool (outside of SynSQLite3.pas/SynDBExplorer).

TSQLRecordNoCaseExtended = class(TSQLRecordNoCase)

Database records with NOCASE collation and JSON_OPTIONS_FAST_EXTENDED variants.
- abstract ancestor, from which you may inherit your own ORM classes.

TSQLRecordCaseSensitiveExtended = class(TSQLRecordCaseSensitive)

Database records with BINARY collation and JSON_OPTIONS_FAST_EXTENDED variants.
- abstract ancestor, from which you may inherit your own ORM classes.

TSQLRecordNoCaseExtendedNoID = class(TSQLRecordNoCaseExtended)

Database records with NOCASE collation and JSON_OPTIONS_FAST_EXTENDED variants, and itoNoIndex4TID option to avoid indexes on TID/T*ID properties.
- abstract ancestor, from which you may inherit your own ORM classes.
class procedure InitializeTable(Server: TSQLRestServer; const FieldName: RawUTF8; Options: TSQLInitializeTableOptions); override;

Overriden method forcing no index creation on TID/T*ID properties

TSQLTableFieldType = record

Store TSQLFieldType and RTTI for a given TSQLTable field

ContentDB: TSQLDBFieldType;

How this field could be stored in a database
- equals ftUnknown if InitFields guessed the field type, or for sftVariant

ContentSize: integer;

The field size in bytes; -1 means not computed yet

ContentType: TSQLFieldType;

The field kind, as in JSON (match TSQLPropInfo.SQLFieldTypeStored)

ContentTypeInfo: pointer;

Used for sftEnumerate, sftSet and sftBlobDynArray fields

TableIndex: integer;

The corresponding index in fQueryTables[]

TSQLTable = class(TObject)

Wrapper to an ORM result table, staticaly stored as UTF-8 text
- contain all result in memory, until destroyed
- first row contains the field names
- following rows contains the data itself
- GetString() can be used in a TDrawString
- will be implemented as TSQLTableJSON for remote access through optimized JSON content

Used for DI-2.1.2 (page 2545).

constructor Create(const aSQL: RawUTF8);

Initialize the result table
- you can optionaly associate the corresponding TSQLRecordClass types, by which the results were computed (it will use RTTI for column typing)

constructor CreateFromTables(const Tables: array of TSQLRecordClass; const aSQL: RawUTF8);

Initialize the result table
- you can associate the corresponding TSQLRecordClass types, by which the results were computed (it will use RTTI for column typing)

constructor CreateWithColumnType(const ColumnTypes: array of TSQLFieldType; const aSQL: RawUTF8);

Initialize the result table
- you can set the expected column types matching the results column layout

destructor Destroy; override;

Free associated memory and owned records
function CalculateFieldLengthMean(var aResult: TIntegerDynArray; FromDisplay: boolean=false): integer;

Get the mean of characters length of all fields
- the character length is for the first line of text only (stop counting at every newline character, i.e. #10 or #13 char)
- return the sum of all mean of character lengths

function DeleteColumnValues(Field: integer): boolean;

Delete the specified Column text from the Table
- don't delete the Column: only delete UTF-8 text in all rows for this field

function DeleteRow(Row: integer): boolean;

Delete the specified data Row from the Table
- only overwrite the internal fResults[] pointers, don't free any memory, nor modify the internal DataSet

function ExpandAsString(Row,Field: integer; Client: TObject; out Text: string; const CustomFormat: string=''): TSQLFieldType;

Read-only access to a particular field value, as VCL text
- Client is one TSQLClient instance (used to display TRecordReference via the associated TSQLModel)
- returns the Field Type
- return generic string Text, i.e. UnicodeString for Delphi 2009+, ready to be displayed to the VCL, for sftEnumerate, sftTimeLog, sftUnixTime/sftUnixMSTime and sftRecord/sftRecordVersion/sftID/sftTID
- returns '' as string Text, if text can be displayed directly with Get*() methods above
- returns '' for other properties kind, if UTF8ToString is nil, or the ready to be displayed value if UTF8ToString event is set (to be used mostly with Language.UTF8ToString)
- CustomFormat can optionally set a custom format string, e.g. '%f' or '%n' or complex FormatFloat()/FormatCurr() syntax (as '#,##0.00') for sftFloat and sftCurrency columns (instead of plain JSON float value), or date/time format as expected by FormatDateTime() for all date/time kind of fields (as sftDateTime, sftDateTimeMS, sftTimeLog, sftModTime, sftCreateTime, sftUnixTime, sftUnixMSTime)

function ExpandAsSynUnicode(Row,Field: integer; Client: TObject; out Text: SynUnicode): TSQLFieldType;

Read-only access to a particular field value, as VCL text
- this method is just a wrapper around ExpandAsString method, returning the content as a SynUnicode string type (i.e. UnicodeString since Delphi 2009, and WideString for non Unicode versions of Delphi)
- it is used by the reporting layers of the framework (e.g. TSQLRibbon.AddToReport)

function Field(const FieldName: RawUTF8): variant; overload;

Read-only access to a particular field value, as a variant
- raise an ESQLTableException if called outside valid Step() sequence
- will call GetVariant() method for appropriate data conversion

function Field(FieldIndex: integer): variant; overload;

Read-only access to a particular field value, as a variant
- raise an ESQLTableException if called outside valid Step() sequence
- will call GetVariant() method for appropriate data conversion
function FieldAsFloat(FieldIndex: Integer): TSynExtended; overload;
   Read-only access to a particular field value, as floating-point value
   - raise an ESQLTableException if called outside valid Step() sequence
   - similar to GetAsFloat() method, but for the current Step

function FieldAsFloat(const FieldName: RawUTF8): TSynExtended; overload;
   Read-only access to a particular field value, as floating-point value
   - raise an ESQLTableException if called outside valid Step() sequence
   - similar to GetAsFloat() method, but for the current Step

function FieldAsInteger(FieldIndex: Integer): Int64; overload;
   Read-only access to a particular field value, as Integer
   - raise an ESQLTableException if called outside valid Step() sequence
   - similar to GetAsInteger() method, but for the current Step

function FieldAsInteger(const FieldName: RawUTF8): Int64; overload;
   Read-only access to a particular field value, as Integer
   - raise an ESQLTableException if called outside valid Step() sequence
   - similar to GetAsInteger() method, but for the current Step

function FieldAsRawUTF8(FieldIndex: Integer): RawUTF8; overload;
   Read-only access to a particular field value, as RawUTF8
   - raise an ESQLTableException if called outside valid Step() sequence
   - similar to GetU() method, but for the current Step

function FieldAsRawUTF8(const FieldName: RawUTF8): RawUTF8; overload;
   Read-only access to a particular field value, as RawUTF8
   - raise an ESQLTableException if called outside valid Step() sequence
   - similar to GetU() method, but for the current Step

function FieldAsString(const FieldName: RawUTF8): String; overload;
   Read-only access to a particular field value, as VCL String
   - raise an ESQLTableException if called outside valid Step() sequence
   - similar to GetString() method, but for the current Step

function FieldAsString(FieldIndex: Integer): String; overload;
   Read-only access to a particular field value, as VCL String
   - raise an ESQLTableException if called outside valid Step() sequence
   - similar to GetString() method, but for the current Step

function FieldBuffer(FieldIndex: Integer): PUTF8Char; overload;
   Read-only access to a particular field value, as UTF-8 encoded buffer
   - raise an ESQLTableException if called outside valid Step() sequence
   - similar to Get() method, but for the current Step

function FieldBuffer(const FieldName: RawUTF8): PUTF8Char; overload;
   Read-only access to a particular field value, as UTF-8 encoded buffer
   - raise an ESQLTableException if called outside valid Step() sequence
   - similar to Get() method, but for the current Step

function FieldIndex(const FieldName: RawUTF8): integer; overload;
   Get the Field index of a FieldName
   - return -1 if not found, index (0..FieldCount-1) if found
function FieldIndex(const FieldNames: array of RawUTF8; const FieldIndexes: array of PInteger): integer; overload;

Get the Field indexes of several Field names
- could be used to speed-up field access in a TSQLTable loop, avoiding a FieldIndex(aFieldName) lookup for each value
- returns the number of matching Field names
- set -1 in FieldIndexes[]^ if not found, index (0..FieldCount-1) if found

function FieldIndex(const FieldName: RawUTF8): integer; overload;

Get the Field index of a FieldName
- return -1 if not found, index (0..FieldCount-1) if found

function FieldIndexExisting(const FieldName: RawUTF8): integer; overload;

Get the Field index of a FieldName
- raise an ESQLTableException if not found, index (0..FieldCount-1) if found

function FieldLengthMax(Field: integer; NeverReturnsZero: boolean=false): cardinal;

Get the maximum number of characters of this field

function FieldLengthMean(Field: integer): cardinal;

Get the mean of characters length of this field
- the character length is for the first line of text only (stop counting at every newline character, i.e. #10 or #13 char)
- very fast: calculated only once for all fields

function FieldLengthMeanSum: cardinal;

Get the sum of all mean of characters length of all fields
- very fast: calculated only once for all fields

function FieldNames: TRawUTF8DynArray;

Retrieve all field names as a RawUTF8 dynamic array

function FieldTable(Field: integer): TSQLRecordClass;

Get the record class (i.e. the table) associated to a field
- is nil if this table has no QueryTables property
- very fast: calculated only once for all fields

function FieldType(Field: integer): TSQLFieldType; overload;

Guess the field type from first non null data row
- if QueryTables[] are set, exact field type and enumerate TypeInfo() is retrieved from the Delphi RTTI; otherwise, get from the cells content
- return sftUnknown is all data fields are null
- sftBlob is returned if the field is encoded as SQLite3 BLOB literals (X'53514C697465' e.g.)
- since TSQLTable data is PUTF8Char, string type is sftUTF8Text only
function FieldType(Field: integer; out FieldTypeInfo: PSQLTableFieldType): TSQLFieldType; overload;

Guess the field type from first non null data row
- if QueryTables[] are set, exact field type and (enumerate) TypeInfo() is retrieved from the Delphi RTTI; otherwise, get from the cells content
- return sftUnknown is all data fields are null
- sftBlob is returned if the field is encoded as SQLite3 BLOB literals (X'53514C697465' e.g.)
- since TSQTable data is PUTF8Char, string type is sftUTF8Text only

function FieldValue(const FieldName: RawUTF8; Row: integer): PUTF8Char;

Get the Field content (encoded as UTF-8 text) from a property name
- return nil if not found

function Get(Row: integer; const FieldName: RawUTF8): PUTF8Char; overload;

Read-only access to a particular field value, as UTF-8 encoded buffer
- points to memory buffer allocated by Init()

function Get(Row, Field: integer): PUTF8Char; overload;

Read-only access to a particular field value, as UTF-8 encoded buffer
- if Row and Fields are correct, returns a pointer to the UTF-8 buffer, or nil if the corresponding JSON was null or ""
- if Row and Fields are not correct, returns nil

function GetA(Row, Field: integer): WinAnsiString;

Read-only access to a particular field value, as Win Ansi text

function GetAsCurrency(Row, Field: integer): currency; overload;

Read-only access to a particular field value, as currency value

function GetAsCurrency(Row: integer; const FieldName: RawUTF8): currency; overload;

Read-only access to a particular field value, as currency value

function GetAsDateTime(Row, Field: integer): TDateTime; overload;

Read-only access to a particular field value, as TDateTime value

function GetAsDateTime(Row: integer; const FieldName: RawUTF8): TDateTime;

Read-only access to a particular field value, as TDateTime value
- sftDateTime/sftDateTimeMS will be converted from ISO-8601 text
- sftTimeLog, sftModTime, sftCreateTime will expect the content to be encoded as a TTimeLog Int64 value - as sftInteger may have been identified by TSQTable.InitFieldTypes
- sftUnixTime/sftUnixMSTime field will call UnixTimeToDateTime/UnixMSTimeToDateTime
- for sftTimeLog, sftModTime, sftCreateTime or sftUnixTime fields, you may have to force the column type, since it may be identified as sftInteger or sftCurrency by default from its JSON number content, e.g. via:
  aTable.SetFieldType('FieldName', sftModTime);
- sftCurrency, sftFloat will return the corresponding double value
- any other types will try to convert ISO-8601 text }

function GetAsFloat(Row, Field: integer): TSynExtended; overload;

Read-only access to a particular field value, as extended value
function GetAsFloat(Row: integer; const FieldName: RawUTF8): TSynExtended; overload;
  Read-only access to a particular field value, as extended value

function GetAsInt64(Row: integer; const FieldName: RawUTF8): Int64; overload;
  Read-only access to a particular field value, as Int64 value

function GetAsInt64(Row,Field: integer): Int64; overload;
  Read-only access to a particular field value, as Int64 value

function GetAsInteger(Row: integer; const FieldName: RawUTF8): integer; overload;
  Read-only access to a particular field value, as integer value

function GetAsInteger(Row,Field: integer): integer; overload;
  Read-only access to a particular field value, as integer value

function GetBlob(Row,Field: integer): TSQLRawBlob;
  Read-only access to a particular Blob value
  - a new TSQLRawBlob is created
  - Blob data is converted from SQLite3 BLOB literals ('X'53514C697465' e.g.) or Base-64 encoded content ('\uFF00base64encodedbinary')
  - preferred manner is to directly use REST protocol to retrieve a blob field

function GetBytes(Row,Field: integer): TBytes;
  Read-only access to a particular Blob value
  - a new TBytes is created
  - Blob data is converted from SQLite3 BLOB literals ('X'53514C697465' e.g.) or Base-64 encoded content ('\uFF00base64encodedbinary')
  - preferred manner is to directly use REST protocol to retrieve a blob field

function GetCaption(Row,Field: integer): string;
  Read-only access to a particular field value, ready to be displayed
  - mostly used with Row=0, i.e. to get a display value from a field name
  - use "string" type, i.e. UnicodeString for Delphi 2009+
  - value is first un-camel-cased: 'OnLine' value will return 'On line' e.g.
  - then System.LoadResStringTranslate() is called if available

function GetCSVValues(Tab: boolean; CommaSep: AnsiChar=','; AddBOM: boolean=false; RowFirst: integer=0; RowLast: integer=0): RawUTF8; overload;
  Save the table as CSV format, into a string variable
  - if Tab=TRUE, will use TAB instead of ',' between columns
  - you can customize the ',' separator - use e.g. the global ListSeparator variable (from SysUtils) to reflect the current system definition (some country use ',' as decimal separator, for instance our "douce France")
  - AddBOM will add a UTF-8 Byte Order Mark at the beginning of the content

function GetDateTime(Row,Field: integer): TDateTime;
  Read-only access to a particular DateTime field value
  - expect SQLite3 TEXT field in ISO 8601 'YYYYMMDD hhmmss' or 'YYYY-MM-DD hh:mm:ss' format
function GetHtmlTable(const Header: RawUTF8='\n<head><style>table,th,td' + '{border: 1px solid black;border-collapse: collapse;}th,td{padding: 5px;' + 'font-family: sans-serif;}</style></head>' + '@#10): RawUTF8; overload;

Save the table as a <html><body><table> </table></body></html> content

function GetJSONValues(Expand: boolean; IDBinarySize: integer=0; BufferSize: integer=0): RawUTF8; overload;

Same as the overloaded method, but returning result in a RawUTF8

Used for DI-2.1.2 (page 2545).

function GetMSRowSetValues: RawUTF8; overload;

Save the table in 'schemas-microsoft-com:rowset' XML format
- this format is used by ADOODB.recordset, easily consumed by MS apps
- see @https://synopse.info/forum/viewtopic.php?id=11691#p11691

function GetODSDocument(withColumnTypes: boolean=false): RawByteString;

Save the table in Open Document Spreadsheet compressed format
- this is a set of XML files compressed in a zip container
- this method will return the raw binary buffer of the file
- see @https://synopse.info/forum/viewtopic.php?id=2133

function GetRowLengths(Field: integer; var LenStore: TSynTempBuffer): integer;

Get all values lengths for a specified field into a PIntegerArray
- returns the total length as result, and fill LenStore with all rows individual lengths using StrLen()
- caller should eventually call LenStore.Done to release any temp memory
- returns 0 if Field is invalid or no data is stored in this TSQLTable - don't call LenStore.Done in this case

function GetRowValues(Field: integer; const Sep: RawUTF8='\', const Head: RawUTF8=''; const Trail: RawUTF8=''): RawUTF8; overload;

Get all values for a specified field as CSV
- don't perform any conversion, but create a CSV from raw PUTF8Char data

function GetRowValues(Field: integer; out Values: TRawUTF8DynArray): integer;

overload;

Get all values for a specified field into a dynamic RawUTF8 array
- don't perform any conversion, but just create an array of raw PUTF8Char data
- returns the number of rows in Values[]

function GetRowValues(Field: integer; out Values: TInt64DynArray): integer;

overload;

Get all values for a specified field into a dynamic Integer array
- returns the number of rows in Values[]

function GetS(Row,Field: integer): shortstring;

Read-only access to a particular field value, as Win Ansi text shortstring

function GetStream(Row,Field: integer): TStream;

Read-only access to a particular Blob value
- a new TCustomMemoryStream is created - caller shall free its instance
- Blob data is converted from SQLite3 BLOB literals (X'53514C697465' e.g.) or Base-64 encoded content ('\uFF00base64encodedbinary')
- preferred manner is to directly use REST protocol to retrieve a blob field
function GetString(Row, Field: integer): string;
   Read-only access to a particular field value, as VCL string text
   - the global UTF8ToString() function will be used for the conversion: for proper i18n handling
   before Delphi 2009, you should use the overloaded method with aUTF8ToString=Language.UTF8ToString

function GetSynUnicode(Row, Field: integer): SynUnicode;
   Read-only access to a particular field value, as fast Unicode string text
   - SynUnicode is either WideString, either UnicodeString, depending on the Delphi compiler
   revision, to ensure fastest native Unicode process available

function GetTimeLog(Row, Field: integer; Expanded: boolean; FirstTimeChar: AnsiChar = 'T'): RawUTF8;
   Read-only access to a particular TTimeLog field value
   - return the result as TTimeLogBits.Text() Iso-8601 encoded text

function GetU(Row, Field: integer): RawUTF8; overload;
   Read-only access to a particular field value, as RawUTF8 text

function GetU(Row: integer; const FieldName: RawUTF8): RawUTF8; overload;
   Read-only access to a particular field value, as RawUTF8 text

function GetValue(const aLookupFieldName, aLookupValue, aValueFieldName: RawUTF8): variant;
   Read-only access to a particular field, via a lookup field name
   - will call GetVariant() on the corresponding field
   - returns null if the lookup did not have any match

function GetVariant(Row, Field: integer): variant; overload;
   Read-only access to a particular field value, as a Variant
   - text will be stored as RawUTF8 (as varString type)
   - will try to use the most appropriate Variant type for conversion (will use e.g. TDateTime for
     sftDateTime, or a TDocVariant for JSON objects in a sftVariant column) - so you should better set
     the exact field types (e.g. from ORM) before calling this method

function GetW(Row, Field: integer): RawUnicode;
   Read-only access to a particular field value, as UTF-16 Unicode text
   - Raw Unicode is WideChar(zero) terminated
   - its content is allocated to contain all WideChars (not trimed to 255, like GetWP() above

function GetWP(Row, Field: integer; Dest: PWideChar; MaxDestChars: cardinal): integer;
   Fill a unicode buffer with a particular field value
   - return number of wide characters written in Dest^

function IDColumnHiddenValue(Row: integer): TID;
   Return the (previously hidden) ID value, 0 on error

function IDColumnHide: boolean;
   If the ID column is available, hides it from fResults[]
   - useful for simplier UI, with a hidden ID field
   - use IDColumnHiddenValue() to get the ID of a specific row
   - return true is ID was succesfully hidden, false if not possible
function LengthW(Row, Field: integer): integer;
    Widechar length (UTF-8 decoded as UTF-16) of a particular field value
    - could be used with VCL's UnicodeString, or for Windows API

function NewRecord(RecordType: TSQLRecordClass=nil): TSQLRecord;
    Create a new TSQLRecord instance for a specific Table
    - a void TSQLRecord instance is created, ready to be filled
    - use the specified TSQLRecord class or create one instance of the first associated record class
      (from internal QueryTables[])
    - the returned records will be managed by this TSQLTable: they will be freed when the
      TSQLTable is destroyed: you don't need to make a try..finally..Free..end block with them

function QueryRecordType: TSQLRecordClass;
    Retrieve QueryTables[0], if existing

function RowFromID(aID: TID; aNotFoundMinusOne: boolean=false): integer;
    Get the Row index corresponding to a specified ID
    - return the Row number, from 1 to RowCount
    - return RowCount (last row index) if this ID was not found or no ID field is available, unless
      aNotFoundMinusOne is set, and then -1 is returned

function SearchFieldEquals(const Value: RawUTF8; FieldIndex: integer; StartRow: integer=1; CaseSensitive: boolean=false): integer; overload;
    Search for a value inside the raw table data, using UTF8IComp/StrComp()
    - returns 0 if not found, or the matching Row number otherwise

function SearchFieldEquals(Value: PUTF8Char; FieldIndex: integer; StartRow: integer=1; CaseSensitive: boolean=false): integer; overload;
    Search for a value inside the raw table data, using UTF8IComp/StrComp()
    - returns 0 if not found, or the matching Row number otherwise

function SearchFieldIdemPChar(const Value: RawUTF8; FieldIndex: integer; StartRow: integer=1): integer;
    Search for a value inside the raw table data, using IdemPChar()
    - returns 0 if not found, or the matching Row number otherwise

function SearchFieldSorted(const Value: RawUTF8; FieldIndex: integer; CustomCompare: TUTF8Compare=nil): integer; overload;
    Search for a value using O(log(n)) binary search of a sorted field
    - here the content should have been previously sorted via Sort(), or CustomCompare should be
      defined, otherwise the SearchFieldEquals() slower O(n) method is called
    - returns 0 if not found, or the matching Row number otherwise

function SearchFieldSorted(Value: PUTF8Char; FieldIndex: integer; CustomCompare: TUTF8Compare=nil): integer; overload;
    Search for a value using O(log(n)) binary search of a sorted field
    - here the content should have been previously sorted via Sort(), or CustomCompare should be
      defined, otherwise the SearchFieldEquals() slower O(n) method is called
    - returns 0 if not found, or the matching Row number otherwise
function SearchValue(const UpperValue: RawUTF8; StartRow, FieldIndex: integer; Client: TObject; Lang: TSynSoundExPronunciation=sndxNone; UnicodeComparison: boolean=false): integer; overload;

Search a text value inside the table data in a specified field
- the text value must already be uppercased 7-bits ANSI encoded
- return the Row on success, 0 on error
- search only in the content of FieldIndex data
- you can specify a Soundex pronunciation to use, or leave as sndxNone for standard case insensitive character match; aUpperValue can optional indicate a Soundex search, by predeceding the searched text with % for English, % for French or %%% for Spanish (only works with WinAnsi char set - i.e. code page 1252)
- if UnicodeComparison is set to TRUE, search will use low-level Windows API for Unicode-level conversion - it will be much slower, but accurate for the whole range of UTF-8 encoding
- if UnicodeComparison is left to FALSE, UTF-8 decoding will be done only if necessary: it will work only with standard western-occidental alphabet (i.e. WinAnsi - code page 1252), but it will be very fast

function SearchValue(const UpperValue: RawUTF8; StartRow: integer; FieldIndex: PInteger; Client: TObject; Lang: TSynSoundExPronunciation=sndxNone; UnicodeComparison: boolean=false): integer; overload;

Search a text value inside the table data in all fields
- the text value must already be uppercased 7-bits ANSI encoded
- return the Row on success, 0 on error
- search on all fields, returning field found in FieldIndex (if not nil)
- you can specify a Soundex pronunciation to use, or leave as sndxNone for standard case insensitive character match; aUpperValue can optional indicate a Soundex search, by predeceding the searched text with % for English, % for French or %%% for Spanish (only works with WinAnsi char set - i.e. code page 1252)
- if UnicodeComparison is set to TRUE, search will use low-level Windows API for Unicode-level conversion - it will be much slower, but accurate for the whole range of UTF-8 encoding
- if UnicodeComparison is left to FALSE, UTF-8 decoding will be done only if necessary: it will work only with standard western-occidental alphabet (i.e. WinAnsi - code page 1252), but it will be very fast

function SortCompare(Field: integer): TUTF8Compare;

Get the appropriate Sort comparison function for a field, nil if not available (bad field index or field is blob)
- field type is guessed from first data row
function Step(SeekFirst: boolean=false; RowVariant: PVariant=ni\l): boolean;

After a TSQLTable has been initialized, this method can be called one or more times to iterate through all data rows
- you shall call this method before calling FieldBuffer()/Field() methods
- return TRUE on success, with data ready to be retrieved by Field*
- return FALSE if no more row is available (i.e. exceeded RowCount)
- if SeekFirst is TRUE, will put the cursor on the first row of results, otherwise, it will fetch one row of data, to be called within a loop
- you can specify a variant instance (e.g. allocated on the stack) in optional RowVariant parameter, to access field values using late binding
- typical use may be:
  while TableCustomers.Step do
    writeln(Field('name'));
- or, when using a variant and late-binding:
  var customer: variant;
  ...
  while TableCustomers.Step(false,@customer) do
    writeln(customer.Name);

function ToObjArray(var ObjArray; RecordType: TSQLRecordClass=ni\l): boolean;

Fill an existing T*ObjArray variable with TSQLRecord instances corresponding to this TSQLTable result set
- use the specified TSQLRecord class or create instances of the first associated record class (from internal QueryTables[])  
- returns TRUE on success (even if ObjArray=[]), FALSE on error

function ToObjectList(RecordType: TSQLRecordClass=ni\l): TObjectList; overload;

Create a TObjectList with TSQLRecord instances corresponding to this TSQLTable result set
- use the specified TSQLRecord class or create instances of the first associated record class (from internal QueryTables[])  
- always returns an instance, even if the TSQLTable is nil or void

procedure Assign(source: TSQLTable);

Copy the parameters of a TSQLTable into this instance
- the fResults remain in the source TSQLTable: source TSQLTable has not to be destroyed before this TSQLTable

procedure FieldIndexExisting(const FieldNames: array of RawUTF8; const FieldIndexes: array of PInteger); overload;

Get the Field indexes of several Field names
- raise an ESQLTableException if not found
- set FieldIndexes[]^ to the index (0..FieldCount-1) if found
- could be used to speed-up field access in a TSQLTable loop, avoiding a FieldIndex(aFieldName) lookup for each value, as such:

```pascal
list := TSQLTableJSON.Create('',pointer(json),length(json));
list.FieldIndexExisting( [ 'FirstName', 'LastName', 'YearOfBirth', 'YearOfDeath', 'RowID', 'Data' ],
[@FirstName, @LastName, @YearOfBirth, @YearOfDeath, @RowID, @Data]);
for i := 1 to list.RowCount do begin
  Check(list.Get(i,FirstName)<nil);
  Check(list.Get(i,LastName)<nil);
  Check(list.GetAsInteger(i,YearOfBirth)<10000);
```
procedure FieldLengthMeanIncrease(aField, aIncrease: integer);

Increase a particular Field Length Mean value
- to be used to customize the field appearence (e.g. for adding of left checkbox for Marked[] fields)

procedure GetAsVariant(row, field: integer; out value: variant;
expandTimeLogAsText, expandEnumsAsText, expandHugeIDAsUniqueIdentifier: boolean;
options: TDocVariantOptions=JSON_OPTIONS_FAST);

Retrieve a field value in a variant
- returns null if the row/field is incorrect
- expand* methods will allow to return human-friendly representations

procedure GetCSVValues(Dest: TStream; Tab: boolean; CommaSep: AnsiChar='';
AddBOM: boolean=false; RowFirst: integer=0; RowLast: integer=0);

Save the table as CSV format, into a stream
- if Tab=TRUE, will use TAB instead of ',' between columns
- you can customize the ',', separator - use e.g. the global ListSeparator variable (from SysUtils) to reflect the current system definition (some country use ',' as decimal separator, for instance our "douce France")
- AddBOM will add a UTF-8 Byte Order Mark at the beginning of the content

procedure GetHtmlTable(Dest: TTextWriter); overload;

Append the table content as a HTML <table> ... </table>

procedure GetJSONValues(JSON: TStream; Expand: boolean; RowFirst: integer=0;
RowLast: integer=0; IDBinarySize: integer=0);

Same as the overloaded method, but appending an array to a TStream
- Used for DI-2.1.2 (page 2545).

procedure GetMSRowSetValues(Dest: TStream; RowFirst,RowLast: integer);

Save the table in 'schemas-microsoft-com:rowset' XML format
- this format is used by ADODB.recordset, easily consumed by MS apps
- see @https://synopse.info/forum/viewtopic.php?id=11691#p11691

procedures GetJSONValues(W: TJSONWriter; RowFirst: integer=0; RowLast: integer=0;
IDBinarySize: integer=0); overload;

{ ifdef HASINLINE }inline{ endif } won't reset docarray as required save the table values in JSON format
- JSON data is added to TJSONWriter, with UTF-8 encoding, and not flushed
- if Expand is true, JSON data is an array of objects, for direct use with any Ajax or .NET client:
  {"col1":val11,"col2":"val12"},{"col1":val21,... }
- if W.Expand is false, JSON data is serialized (used in TSQLTableJSON)
  { "fieldCount":1,"values":["col1","col2",val11,"val12",val21,... ] }

- RowFirst and RowLast can be used to ask for a specified row extent of the returned data (by default, all rows are retrieved)
- IDBinarySize will force the ID field to be stored as hexadecimal text

{ ifdef HASINLINE }inline{ endif } won't reset docarray as required save the table values in JSON format
- JSON data is added to TJSONWriter, with UTF-8 encoding, and not flushed
- if Expand is true, JSON data is an array of objects, for direct use with any Ajax or .NET client:
  {"col1":val11,"col2":"val12"},{"col1":val21,... }
- if W.Expand is false, JSON data is serialized (used in TSQLTableJSON)
  { "fieldCount":1,"values":["col1","col2",val11,"val12",val21,... ] }

- RowFirst and RowLast can be used to ask for a specified row extent of the returned data (by default, all rows are retrieved)
- IDBinarySize will force the ID field to be stored as hexadecimal text

- Used for DI-2.1.2 (page 2545).

procedure GetMSRowSetValues(Dest: TStream; RowFirst,RowLast: integer); overload;

Save the table in 'schemas-microsoft-com:rowset' XML format
- this format is used by ADODB.recordset, easily consumed by MS apps
- see @https://synopse.info/forum/viewtopic.php?id=11691#p11691
procedure GetVariant(Row, Field: integer; var result: variant); overload;

Read-only access to a particular field value, as a Variant
- text will be stored as RawUTF8 (as varString type)
- will try to use the most appropriate Variant type for conversion (will use e.g. TDateTime for sftDateTime, or a TDocVariant for JSON objects in a sftVariant column) - so you should set the exact field types (e.g. from ORM) before calling this method

procedure IDArrayFromBits(const Bits; var IDs: TIDDynArray);
Get all IDs where individual bit in Bits are set

procedure IDArrayToBits(var Bits; var IDs: TIDDynArray);
Get all individual bit in Bits corresponding to the supplied IDs
- warning: IDs integer array will be sorted within this method call

procedure IDColumnHiddenValues(var IDs: TIDDynArray);
Return all (previously hidden) ID values

procedure SetFieldLengthMean(const Lengths: array of cardinal);
Force the mean of characters length for every field
- expect as many parameters as fields in this table
- override internal fFieldLengthMean[] and fFieldLengthMeanSum values

procedure SetFieldType(const FieldName: RawUTF8; FieldType: TSQLFieldType; FieldTypeInfo: pointer=nil; FieldSize: integer=-1); overload;
Set the exact type of a given field
- by default, column types and sizes will be retrieved from JSON content from first row, or all rows if FieldTypeIntegerDetectionOnAllRows is set
- you can define a specific type for a given column, and optionally a maximum column size
- FieldTypeInfo can be specified for sets or enumerations, as such:
aTable.SetFieldType('Sample', sftEnumerate, TypeInfo(TEnumSample));
aTable.SetFieldType('Samples', sftSet, TypeInfo(TSetSamples));

procedure SetFieldTypes(const DBTypes: TSQLDBFieldTypeDynArray);
Set the exact type of all fields, from the DB-like information

procedure SortBitsFirst(var Bits);
Sort result Rows, according to the Bits set to 1 first
procedure SortFields(const Fields: array of integer; const Asc: array of boolean; const CustomCompare: array of TUTF8Compare); overload;

Sort resultRows, according to some specific fields
- is able to make multi-field sort
- both Fields[] and Asc[] arrays should have the same count, otherwise default Asc[]=true value will be assumed
- set any Fields[]= -1 to identify the ID column (even if is hidden)
- if CustomCompare=[], which use the default comparison function for the field type, unless you set as many custom comparison function items as in the Fields[] and Asc[] parameters

procedure SortFields(const FieldName: RawUTF8; Asc: boolean=true; PCurrentRow: PInteger=nil; FieldType: TSQLFieldType=sftUnknown; CustomCompare: TUTF8Compare=nil); overload;

Sort resultRows, according to a specific field
- overloaded method allowing to specify the field by its name

procedure SortFields(Field: integer; Asc: boolean=true; PCurrentRow: PInteger=nil; FieldType: TSQLFieldType=sftUnknown; CustomCompare: TUTF8Compare=nil); overload;

Sort resultRows, according to a specific field
- default is sorting by ascending order (Asc=true)
- you can specify a Row index to be updated during the sort in PCurrentRow
- sort is very fast, even for huge tables (more faster than any indexed SQL query): 500,000 rows are sorted instantly
- this optimized sort implementation does the comparison first by the designed field, and, if the field value is identical, the ID value is used (it will therefore sort by time all identical values)

procedure ToDocVariant(Row: integer; out doc: variant; options: TDocVariantOptions=JSON_OPTIONS_FAST; expandTimeLogAsText: boolean=false; expandEnumsAsText: boolean=false; expandHugeIDAsUniqueIdentifier: boolean=false); overload;

Retrieve a row value as a variant, ready to be accessed via late-binding
- Row parameter numbering starts from 1 to RowCount
- this method will return a TDocVariant containing a copy of all field values of this row, uncoupled to the TSQLTable instance life time
- expand* methods will allow to return human-friendly representations

procedure ToDocVariant(out docs: TVariantDynArray; readonly: boolean); overload;

Retrieve all row values as a dynamic array of variants, ready to be accessed via late-binding
- if readonly is TRUE, will contain an array of TSQLTableRowVariant, which will point directly to the TSQLTable, which should remain allocated
- if readonly is FALSE, will contain an array of TDocVariant, containing a copy of all field values of this row, uncoupled to the TSQLTable instance
- readonly=TRUE is faster to allocate (around 4 times for 10,000 rows), but may be slightly slower to access than readonly=FALSE, if all values are likely be accessed later in the process
procedure ToDocVariant(out docarray: variant; readonly: boolean); overload;
  Retrieve all row values as a TDocVariant of kind dvArray, ready to be accessed via late-binding
  - if readonly is TRUE, will contain an array of TSQLTableRowVariant, which will point directly to
    the TSQLTable, which should remain allocated
  - if readonly is FALSE, will contain an array of TDocVariant, containing a copy of all field values of
    this row, uncoupled to the TSQLTable instance
  - readonly=TRUE is faster to allocate (around 4 times for 10,000 rows), but may be slightly
    slower to access than readonly=FALSE, if all values are likely be accessed later in the process

procedure ToObjectList(DestList: TObjectList; RecordType: TSQLRecordClass=nil); overload;
  Fill an existing TObjectList with TSQLRecord instances corresponding to this TSQLTable result set
  - use the specified TSQLRecord class or create instances of the first associated record class (from
    internal QueryTables[])
property RowCount: integer read GetRowCount;

Read-only access to the number of data Rows in this table
- first row contains field name
- then 1..RowCount rows contain the data itself
- safely returns 0 if the TSQLTable instance is nil

property StepRow: integer read fStepRow;

Read-only access to the current Row number, after a Step() call
- contains 0 if accessed outside valid Step() sequence call
- contains 1..RowCount after a valid Step() iteration

TSQLTableRowVariantData = packed record
Memory structure used for our TSQLTableRowVariant custom variant type used to have direct access to TSQLTable content
- the associated TSQLTable must stay allocated as long as this variant is used, otherwise random GPF issues may occur

VRow: integer;
The row number corresponding to this value
- equals -1 if should follow StepRow property value

VTable: TSQLTable;
Reference to the associated TSQLTable

VType: TVarType;
The custom variant type registered number

TSQLTableRowVariant = class(TSynInvokeableVariantType)
A custom variant type used to have direct access to TSQLTable content
- use TSQLTable.Step(.,.@Data) method to initialize such a Variant
- the variant members/fields are read-only by design
- the associated TSQLTable must stay allocated as long as this variant is used, otherwise random GPF issues may occur

procedure Cast(var Dest: TVarData; const Source: TVarData); override;
Handle type conversion to string

procedure CastTo(var Dest: TVarData; const Source: TVarData; const AVarType: TVarType); override;
Handle type conversion to string

procedure ToJSON(W: TTextWriter; const Value: TVarVariant; Escape: TTextWriterKind); override;
Customization of variant into JSON serialization
TObjectVariant = class(TSynInvokeableVariantType)

A custom variant type used to have direct access to object published properties
- TObjectVariant provides lazy-loading to object properties from a Variant variable - which may be used with SynMustache or with late-binding
- Warning: this custom variant is just a wrapper around an existing TObject instance, which should remain available as long as the variant is used
- If you want a per-representation stateless variant, use ObjectToVariant() which convert all properties into a TDocVariant, so may use more resource

class procedure New(var V: Variant; Obj: TObject);

Initialize a new custom variant instance, wrapping the specified object
- Warning: this custom variant is just a wrapper around an existing TObject instance, which should remain available as long as the variant is used

procedure ToJSON(W: TTextWriter; const Value: variant; Escape: TTextWriterKind); override;

Will perform proper JSON serialization calling W.WriteObject()

TSQLTableJSON = class(TSQLTable)

Store a read-only ORM result table from a JSON message
- The JSON data is parsed and unescaped in-place, to enhanced performance and reduce resource consumption (mainly memory/heap fragmentation)
- Is used by the ORM for TSQLRecord.FillPrepare/FillOne methods for fast access to individual object values

Used for DI-2.1.1 (page 2543), DI-2.1.2 (page 2545).

creator Create(const aSQL, aJSON: RawUTF8); reintroduce; overload;

Create the result table from a JSON-formated Data message
- The JSON data is parsed and formatted in-place, after having been copied in the protected fPrivateCopy variable

Used for DI-2.1.2 (page 2545).

creator Create(const aSQL: RawUTF8; JSONBuffer: PUTF8Char; JSONBufferLen: integer); reintroduce; overload;

Create the result table from a JSON-formated Data message
- The JSON data is parsed and formatted in-place
- Please note that the supplied JSON buffer content will be changed: if you want to reuse this JSON content, you shall make a private copy before calling this constructor and you shall NOT release the corresponding variable (fResults/JSONResults[] will point inside this memory buffer): use instead the overloaded Create constructor expecting a const aJSON:RawUTF8 parameter to allocate and hold a private copy of the data

Used for DI-2.1.2 (page 2545).
**constructor** CreateFromTables(const Tables: array of TSQLRecordClass; const aSQL, aJSON: RawUTF8; reintroduce; overload;

Create the result table from a JSON-formatted Data message
- you can specify a set of TSQLRecord classes which will be used to retrieve the column exact type information
- the JSON data is parsed and formatted in-place, after having been copied in the protected fPrivateCopy variable

**constructor** CreateFromTables(const Tables: array of TSQLRecordClass; const aSQL: RawUTF8; JSONBuffer: PUTF8Char; JSONBufferLen: integer); reintroduce; overload;

Create the result table from a JSON-formatted Data message
- the JSON data is parsed and formatted in-place
- you can specify a set of TSQLRecord classes which will be used to retrieve the column exact type information
- please note that the supplied JSON buffer content will be changed

**constructor** CreateWithColumnTypes(const ColumnTypes: array of TSQLFieldType; const aSQL, aJSON: RawUTF8); reintroduce; overload;

Initialize the result table from a JSON-formatted Data message
- you can set the expected column types matching the results column layout
- the JSON data is parsed and formatted in-place, after having been copied in the protected fPrivateCopy variable

**constructor** CreateWithColumnTypes(const ColumnTypes: array of TSQLFieldType; const aSQL: RawUTF8; JSONBuffer: PUTF8Char; JSONBufferLen: integer); reintroduce; overload;

Initialize the result table from a JSON-formatted Data message
- you can set the expected column types matching the results column layout
- the JSON data is parsed and formatted in-place

**function** UpdateFrom(const aJSON: RawUTF8; var Refreshed: boolean; PCurrentRow: PInteger): boolean;

Update the result table content from a JSON-formatted Data message
- return true on parsing success, false if no valid JSON data was found
- set Refreshed to true if the content changed
- update all content fields (fResults[], fRowCount, fFieldCount, etc...)
- call SortFields() or IDCOLUMNHIDE if was already done for this TSQLTable
- the conversion into PUTF8CharArray is made in-place and is very fast (only one memory buffer is allocated for the whole data)

Used for DI-2.1.2 (page 2545).

**property** PrivateInternalCopy: RawUTF8 read fPrivateCopy;

The private copy of the processed data buffer
- available e.g. for Create constructor using aJSON parameter, or after the UpdateFrom() process
- 16 more bytes will be allocated, to allow e.g. proper SSE4.2 process
- this buffer is not to be access directly: this won't be a valid JSON content, but a processed buffer, on which fResults[] elements point to - it will contain unescaped text and numerical values, ending with #0
TSQLTableWritable = class(TSQLTableJSON)
  Store a writable ORM result table, optionally read from a JSON message
- in respect to TSQLTableJSON, this class allows to modify field values, and add some new fields on
  the fly, even joined from another TSQLTable

function AddField(const FieldName: RawUTF8; FieldType: TSQLFieldType;
  FieldTypeInfo: pointer=nil; FieldSize: integer=-1): integer; overload;
  Define a new field to be stored in this table
- returns the internal index of the newly created field

function AddField(const FieldName: RawUTF8): integer; overload;
  Define a new field to be stored in this table
- returns the internal index of the newly created field

function AddField(const FieldName: RawUTF8; FieldTable: TSQLRecordClass;
  const FieldTableName: RawUTF8=''): integer; overload;
  Define a new field to be stored in this table
- returns the internal index of the newly created field

procedure Join(From: TSQLTable; const FromKeyField, KeyField: RawUTF8);
  Append/merge data from a secondary TSQLTable
- you should specify the primary keys on which the data rows are merged
- merged data will point to From.fResults[] content: so the From instance should remain
  available as long as you use this TSQLTableWritable
- warning: will call From.SortFields(FromKeyField) for faster process

procedure Update(Row: integer; const FieldName, Value: RawUTF8); overload;
  Modify a field value in-place, using a RawUTF8 text value

procedure Update(Row,Field: integer; const Value: RawUTF8); overload;
  Modify a field value in-place, using a RawUTF8 text value

procedure Update(Row: integer; const FieldName: RawUTF8; const Value: variant);
  overload;
  Modify a field value in-place, using a variant value

procedure Update(Row,Field: integer; const Value: variant); overload;
  Modify a field value in-place, using a variant value

property NewValuesCount: integer read fNewValuesCount;
  How many values have been written via Update() overloaded methods
- is not used if NewValuesInterning was defined

property NewValuesInterning: TRawUTF8Interning read fNewValuesInterning write
  fNewValuesInterning;
  Optionaly de duplicate Update() values

TSQLLocks = object(TObject)
  Used to store the locked record list, in a specified table
- the maximum count of the locked list if fixed to 512 by default, which seems correct for common
  usage
Count: integer;
   *The number of locked records stored in this object*

IDs: TIDDynArray;
   *Contains the locked record ID*
   - an empty position is marked with 0 after UnLock()

Ticks64s: TInt64DynArray;
   *Contains the time and date of the lock*
   - filled internally by the fast GetTickCount64() function (faster than TDateTime or TSystemTime/GetLocalTime)
   - used to purge to old entries - see PurgeOlderThan() method below

function isLocked(aID: TID): boolean;
   *Return true if a record, specified by its ID, is locked*

function Lock(aID: TID): boolean;
   *Lock a record, specified by its ID*
   - returns true on success, false if was already locked

function UnLock(aID: TID): boolean;
   *Unlock a record, specified by its ID*
   - returns true on success, false if was not already locked

procedure PurgeOlderThan(MinutesFromNow: cardinal=30);
   *Delete all the locked IDs entries, after a specified time*
   - to be used to release locked records if the client crashed
   - default value is 30 minutes, which seems correct for common database usage

TSQLQueryCustom = record
   *Store one custom query parameters*
   - add custom query by using the TSQLRest.QueryAddCustom() method
   - use EnumType^.GetCaption(EnumIndex) to retrieve the caption associated to this custom query

EnumIndex: integer;
   *The associated enumeration index in EnumType*
   - will be used to fill the Operator parameter for the Event call

EnumType: PEnumType;
   *The associated enumeration type*

Event: TSQLQueryEvent;
   *The associated evaluation Event handler*
   - the Operator parameter will be filled with the EnumIndex value

Operators: TSQLQueryOperators;
   *User Interface Query action operators*
TSQLRibbonTabParameters = object(TObject)

Defines the settings for a Tab for User Interface generation
- used in mORMotToolBar.pas unit and TSQLMModel.Create() overloaded method
- is defined as an object and not a record to allow easy inheritance for proper per-application customization - see e.g. FileTables.pas in main demo

AutoRefresh: boolean;

By default, the screens are not refreshed automatically
- but you can enable the auto-refresh feature by setting this property to TRUE, and creating a WM_TIMER message handler for the form, which will handle both WM_TIMER_REFRESH_SCREEN and WM_TIMER_REFRESH_REPORT timers:
  procedure TMainForm.WMRefreshTimer(var Msg: TWMTimer);
  begin
    Ribbon.WMRefreshTimer(Msg);
  end;

CustomCaption: PResStringRec;

The caption of the Tab, to be translated on the screen
- by default, Tab name is taken from TSQLMRecord.Caption(nil) method
- but you can override this value by setting a pointer to a resourcestring

CustomHint: PResStringRec;

The hint type of the Tab, to be translated on the screen
- by default, hint will replace all %s instance by the Tab name, as taken from TSQLMRecord.Caption(nil) method
- but you can override this value by setting a pointer to a resourcestring

EditExpandFieldHints: boolean;

Write hints above field during the edition of this table
- if EditExpandFieldHints is TRUE, the hints are written as text on the dialog, just above the field content; by default, hints are displayed as standard delayed popup when the mouse hover the field editor

EditFieldHints: PResStringRec;

The associated hints to be displayed during the edition of this table
- every field hint must be separated by a '|' character (e.g. 'The First Name|Its Company Name')
- all fields need to be listed in this text resource, even if it won't be displayed on screen (enter a void item like ||)
- you can define some value by setting a pointer to a resourcestring

EditFieldHintsToReport: boolean;

If the default report must contain the edit field hints
- i.e. if the resourcestring pointed by EditFieldHints must be used to display some text above every property value on the reports

EditFieldNameToHideCSV: RawUTF8;

A CSV list of field names to be hidden in both editor and default report
- handy to hide fields containing JSON data or the name of another sftRecord/sftID/sftTID (i.e. TRecordReference/TSQLMRecord props) fields
- list is to be separated by commas (e.g. "RunLogJSON,OrdersJSON" or "ConnectionName")
EditFieldNameWidth: integer;
   The associated field name width (in pixels) to be used for creating the edition dialog for this table

FieldWidth: RawUTF8;
   Displayed field length mean, one char per field (A=1,Z=26)
   - put lowercase character in order to center the field data

Group: integer;
   Tab Group number (index starting at 0)

Layout: TSQLListLayout;
   Layout of the List, below the ribbon

ListWidth: integer;
   Width of the List, in percent of the client area
   - default value (as stated in TSQLRibbonTab.Create) is 30%

NoReport: boolean;
   By default, the detail are displayed as a report (TGDIPages component)
   - set this property to true to customize the details display
   - this property is ignored if Layout is llClient (i.e. details hidden)

OrderFieldIndex: integer;
   Index of field used for displaying order

ReverseOrder: boolean;
   If set, the list is displayed in reverse order (i.e. decreasing)

Select: RawUTF8;
   SQL fields to be displayed on the data lists 'ID,' is always added at the beginning

ShowID: boolean;
   If set, the ID column is shown

Table: TSQLRecordClass;
   The Table associated to this Tab

TSQLRecordVirtual = class(TSQLRecord)
   Parent of all virtual classes
   - you can define a plain TSQLRecord class as virtual if needed  - e.g. inheriting from
   TSQLRecordMany then calling VirtualTableExternalRegister() - but using this class will seal its state
to be virtual

TSQLModelRecordPropertiesSQL = record
   Pre-computed SQL statements for ORM operations for a given TSQLModelRecordProperties
   instance

   InsertSet: RawUTF8;
      All fields, including the ID field, exposed as 'COL1,COL2'
      - to be used e.g. in TSQLVirtualTableExternal.Insert()
SelectAllJoined: RawUTF8;
   The JOINed SQL statement for reading all fields with ID, including nested TSQLRecord
   pre-allocated instances
   - is "" if there is no nested TSQLRecord

SelectAllWithID: RawUTF8;
   The SQL statement for reading all simple fields with ID
   - to be checked if we may safely call EngineList()

SelectAllWithRowID: RawUTF8;
   The SQL statement for reading all simple fields and RowID
   - to be checked if we may safely call EngineList()

TableSimpleFields: array[boolean,boolean] of RawUTF8;
   The simple field names in a SQL SELECT compatible format: 'COL1,COL2' e.g.
   - format is SQL.TableSimpleFields[withID: boolean; withTableName: boolean]
   - returns '*' if no field is of TSQLRawBlob/TSQLRecordMany kind
   - returns 'COL1,COL2' with all COL* set to simple field names if withID is false
   - returns 'ID,COL1,COL2' with all COL* set to simple field names if withID is true
   - returns 'Table.ID,Table.COL1,Table.COL2' if withTableName and withID are true

UpdateSetAll: RawUTF8;
   All updated fields exposed as 'COL1=?,COL2=?'
   - excluding ID (but including TCreateTime fields - as used in TSQLVirtualTableExternal.Update
   method)
   - to be used e.g. for UPDATE statements

UpdateSetSimple: RawUTF8;
   The updated simple fields exposed as 'COL1=?,COL2=?'
   - excluding ID (but including TCreateTime fields - as used in TSQLVirtualTableExternal.Update
   method)
   - to be used e.g. for UPDATE statements

TSQLRecordPropertiesMapping = object(TObject)
   Allow custom field mapping of a TSQLRecord
   - used e.g. for external database process, including SQL generation, as implemented in the
   mORMotDB.pas unit
   - in end user code, mostly MapField/MapFields/Options methods should be used, if needed as a
   fluent chained interface - other lower level methods will be used by the framework internals
   
   function AppendFieldName(FieldIndex: Integer; var Text: RawUTF8): boolean;
   Append a field name to a RawUTF8 Text buffer
   - if FieldIndex=VIRTUAL_TABLE_ROWID_COLUMN (-1), appends RowIDFieldName
   - on error (i.e. if FieldIndex is out of range) will return TRUE
   - otherwise, will return FALSE and append the external field name to Text
function ExternalToInternalIndex(const ExtFieldName: RawUTF8): integer;
  Map an external field name into its internal field index
  - returns the index >=0 in FieldNames[] for a matching external field
  - returns -1 if the field name is RowIDFieldName
  - returns -2 if the field name is not mapped

function ExternalToInternalOrNull(const ExtFieldName: RawUTF8): RawUTF8;
  Map an external field name into its internal field name
  - return '' if the external field name is not RowIDFieldName nor in FieldNames[]

function FieldNameByIndex(FieldIndex: Integer): RawUTF8;
  Return the field name as RawUTF8 value
  - if FieldIndex=VIRTUAL_TABLE_ROWID_COLUMN (-1), appends RowIDFieldName
  - otherwise, will return the external field name

function InternalCSVToExternalCSV(const CSVFieldNames: RawUTF8; const Sep: RawUTF8=''; const SepEnd: RawUTF8=''): RawUTF8;
  Map a CSV list of field names from its internals to its externals values
  - raise an EORMException if any of the supplied field name is not defined in the TSQLRecord as ID or as property (RowIDFieldName or FieldNames[])
  - to be used for a simple CSV (e.g. for INSERT/SELECT statements):
    ExtCSV := InternalCSVToExternalCSV('ID,Name');
  - or for a more complex CSV (e.g. for UPDATE statements):
    ExtCSV := InternalCSVToExternalCSV('ID=?,Name=?','=?,=?');

function InternalToExternal(const FieldName: RawUTF8): RawUTF8;
  Map a field name from its internal name to its external name
  - raise an EORMException if the supplied field name is not defined in the TSQLRecord as ID or a published property

function MapAutoKeywordFields: PSQLRecordPropertiesMapping;
  Call this method to ensure that all fields won't conflict with a SQL keyword for the given database
  - by default, no check is performed: you can use this method to ensure that all field names won’t conflict with a SQL reserved keyword: such fields will be identified and automatically mapped as
    fieldname_{
  - can be used e.g. as
    aModel.Props[TSQLMyExternal].ExternalDB.
    MapField('IntField','ExtField').
    MapAutoKeywordFields;

  - will in fact include the rpmAutoMapKeywordFields flag in Options
  - since it returns a PSQLRecordPropertiesMapping instance, you can chain
    MapField().MapAutoKeywordFields.MapField(); calls to map several fields
function MapField(const InternalName, ExternalName: RawUTF8):
    PSQLRecordPropertiesMapping;

*Add a custom field mapping*
 - will re-compute all needed SQL statements as needed, and initialize fSortedFieldsName[] and fSortedFieldsIndex[] internal sorted arrays
 - can be used e.g. as
  
  ```delphi
  aModel.Props[TSQLMyExternal].ExternalDB.MapField('IntField', 'ExtField');
  ```
 - since it returns a PSQLRecordPropertiesMapping instance, you can chain
  MapField().MapField().MapField(); calls to map several fields

function MapFields(const InternalExternalPairs: array of RawUTF8):
    PSQLRecordPropertiesMapping;

*Add several custom field mappings*
 - can be used e.g. as

  ```delphi
  aModel.Props[TSQLMyExternal].ExternalDB.
       MapFields(['IntField1', 'ExtField1', 'IntField2', 'ExtField2']);
  ```
 - will re-compute all needed SQL statements as needed, and initialize fSortedFieldsName[] and fSortedFieldsIndex[] internal sorted arrays
 - is slightly faster than several chained MapField() calls, since SQL will be computed only once

function SetOptions(aOptions: TSQLRecordPropertiesMappingOptions):
    PSQLRecordPropertiesMapping;

*Specify some advanced options for the field mapping*
 - see TSQLRecordPropertiesMappingOptions for all possibilities
 - can be used e.g. as

  ```delphi
  aModel.Props[TSQLMyExternal].ExternalDB.
       SetOptions([rpmNoCreateMissingTable, rpmNoCreateMissingField]);
  ```
 - since it returns a PSQLRecordPropertiesMapping instance, you can chain
  MapField().SetOptions().MapField(); calls to map several fields

procedure Init(Table: TSQLRecordClass; const MappedTableName: RawUTF8;
    MappedConnection: TObject; AutoComputeSQL: boolean; MappingOptions:
    TSQLRecordPropertiesMappingOptions);

*Initialize the field mapping for a given TSQLRecord*
 - if AutoComputeSQL is true, will pre-compute all needed SQL from the supplied information
 - will left void fSortedFieldsName[] and fSortedFieldsIndex[], to disable custom field mapping

procedure InternalToExternalDynArray(const IntFieldNames: array of RawUTF8; out result: TRawUTF8DynArray; IntFieldIndex: PIntegerDynArray=nil);

*Create a list of external field names, from the internal field names*
 - raise an EORMException if any of the supplied field name is not defined in the TSQLRecord as ID or a published property
 - if IntFieldIndex is set, it will store an array of internal field indexes, i.e. -1 for ID or index in in FieldNames[] for other fields
**property** ConnectionProperties: TObject read fConnectionProperties;

Opaque object used on the Server side to specify e.g. the DB connection
- will define such a generic TObject, to avoid any unnecessary type dependency to other units,
e.g. the SynDB unit in mORMot.pas
- in practice, will be assigned by VirtualTableExternalRegister() to a
  TSQLDBConnectionProperties instance in mORMotDB.pas, or by StaticMongoDBRegister() to a
  TMongoCollection instance, or by TDDDRpositoryRestObjectMapping.Create to its associated
  TSQLRest
- in ORM context, equals nil if the table is internal to SQLite3:
  \[
  \text{if Server.Model.Props[TSQLArticle].ExternalDB.ConnectionProperties=nil then}
  \\text{// this is not an external table, since Init() was not called}
  \]

**property** ExtFieldNames: TRawUTF8DynArray read fExtFieldNames;

The external field names, following fProps.Props.Field[] order
- excluding ID/RowID field, which is stored in RowIDFieldName

**property** ExtFieldNamesUnQuotedSQL: TRawUTF8DynArray read fExtFieldNamesUnQuotedSQL;

The unquoted external field names, following fProps.Props.Field[] order
- excluding ID/RowID field, which is stored in RowIDFieldName
- in respect to ExtFieldNames[], this array will never quote the field name

**property** FieldNamesMatchInternal: TSQLFieldBits read fFieldNamesMatchInternal;

Each bit set, following fProps.Props.Field[]+1 order (i.e. 0=ID, 1=Field[0], ...), indicates that this
external field name has not been mapped

**property** MappingVersion: cardinal read fMappingVersion;

Each time MapField/MapFields is called, this number will increase
- can be used to track mapping changes in real time

**property** Options: TSQLRecordPropertiesMappingOptions read fOptions;

How the mapping process will take place

**property** Properties: TSQLRecordProperties read fProps;

The associated TSQLRecordProperties

**property** RowIDFieldName: RawUTF8 read fRowIDFieldName;

The ID/RowID customized external field name, if any
- is 'ID' by default, since 'RowID' is a reserved column name for some database engines (e.g. Oracle)
- can be customized e.g. via
  aModel.Props[TSQLMyExternal].ExternalDB.MapField('ID', 'ExternalID');

**property** SQL: TSQLModelRecordPropertiesSQL read fSQL;

Pre-computed SQL statements for this external TSQLRecord in this model
- you can use those SQL statements directly with the external engine
- filled if AutoComputeSQL was set to true in Init() method
**property** TableName: RawUTF8 read fTableName;

*Used on the Server side to specify the external DB table name*
- e.g. for including a schema name or an existing table name, with an
  OleDB/MSSQL/Oracle/MySQL/PostgreSQL/Jet.SQLite3 backend
- equals SQLTableName by default (may be overridden e.g. by mORMotDB's
  VirtualTableExternalRegister procedure)

**TSQLModelRecordProperties = class(TObject)**

*ORM properties associated to a TSQLRecord within a given model*
- "stable" / common properties derivated from RTTI are shared in the TSQLRecordProperties instance
- since the same TSQLRecord can be defined in several models, with diverse implementation patterns (e.g. internal in one, external in another), this class is used to regroup all model-specific settings, like SQL pre-generated patterns or external DB properties

ExternalDB: TSQLRecordPropertiesMapping;

*Allow SQL process for one external TSQLRecord in this model*

NoCreateMissingTable: boolean;

*Will by-pass automated table and field creation for this TSQLRecord*
- may be used e.g. when the TSQLRecord is in fact mapped into a View, or is attached as external table and not a real local table

SQL: TSQLModelRecordPropertiesSQL;

*Pre-computed SQL statements for this TSQLRecord in this model*
- those statements will work for internal tables, not for external tables with mapped table or fields names

**constructor** Create(aModel: TSQLModel; aTable: TSQLRecordClass; aKind: TSQLRecordVirtualKind);

*Initialize the ORM properties from the TSQLRecord RTTI and the supplied TSQLModel*

**constructor** CreateFrom(aModel: TSQLModel; aSource: TSQLModelRecordProperties);

*Clone ORM properties from an existing TSQLModelRecordProperties to another model*

**function** SQLFromSelectWhere(const SelectFields, Where: RawUTF8): RawUTF8;

*Compute the SQL statement to be executed for a specific SELECT*
- non simple fields (e.g. BLOBs) will be excluded if SelectFields='*'  
- by default, will return the SELECT statement to be used for internal virtual SQLite3 table - but if ExternalTable is TRUE, then it will compute a SELECT matching ExternalDB settings
procedure FTS4WithoutContent(ContentTable: TSQLRecordClass);
Define if a FTS4 virtual table will not store its content, but will be defined as an "external content" FTS4 table
- see https://www.sqlite.org/fts3.html#section_6_2_2
- the virtual table will be created with content="ContentTableName", and all fields of the FTS4 table
- by design, all fields of the FTS4 table should exist in the source ContentTable - otherwise an exception is raised
- the indexed text will be assigned to the FTS4 table, using triggers generated by TSQLRecordFTS4.InitializeTable at table creation
- note that FTS3 does not support this feature

property Kind: TSQLRecordVirtualKind read fKind write SetKind default rSQLite3;
Define if is a normal table (rSQLite3), an FTS/R-Tree virtual table or a custom TSQLVirtualTable*ID (rCustomForcedID/rCustomAutoID)
- when set, all internal SQL statements will be (re)created, depending of the expected ID/RowID column name expected (i.e. SQLTableSimpleFields[] and SQLSelectAll[] - SQLUpdateSet and SQLInsertSet do not include ID)

property Prop[const PropName: RawUTF8]: TSQLPropInfo read GetProp;
Direct access to a property RTTI information, by name

property Props: TSQLRecordProperties read fProps;
The shared TSQLRecordProperties information of this TSQLRecord
- as retrieved from RTTI

property TableIndex: Integer read fTableIndex;
The table index of this TSQLRecord in the associated Model

TSQLModelRecordReference = record
How a TSQLModel stores a foreign link to be cascaded

CascadeDelete: boolean;
TRUE if this field is a TRecordReferenceToBeDeleted

FieldTable: TSQLRecordClass;
The target TSQLRecordClass of the field

FieldTableIndex: integer;
The target TSQLRecordClass of the field, from its Tables[] index

FieldType: TSQLPropInfo;
The property

TableIndex: integer;
Refers to the source TSQLRecordClass as model Tables[] index
TSQLModel = class(TObject)

A Database Model (in a MVC-driven way), for storing some tables types as TSQLRecord classes
- share this Model between TSQLRest Client and Server
- use this class to access the table properties: do not rely on the low-level database methods (e.g. TSQLDatabase.GetTableNames), since the tables may not exist in the main SQLite3 database, but in-memory or external
- don't modify the order of Tables inside this Model, if you publish some TRecordReference property in any of your tables

constructor Create(CloneFrom: TSQLModel); reintroduce; overload;

Clone an existing Database Model
- all supplied classes won't be redefined as non-virtual: VirtualTableExternalRegister explicit calls are not mandatory here

constructor Create(Owner: TSQLRest; TabParameters: PSQLRibbonTabParameters; TabParametersCount, TabParametersSize: integer; const NonVisibleTables: array of TSQLRecordClass; Actions: PTypeInfo=nil; Events: PTypeInfo=nil; const aRoot: RawUTF8='root'); reintroduce; overload;

Initialize the Database Model from an User Interface parameter structure
- this constructor will reset all supplied classes to be defined as non-virtual (i.e. Kind=rSQLite3): VirtualTableExternalRegister explicit calls are to be made if tables should be managed as external

constructor Create(const Tables: array of TSQLRecordClass; const aRoot: RawUTF8='root'); reintroduce; overload;

Initialize the Database Model
- set the Tables to be associated with this Model, as TSQLRecord classes
- set the optional Root URI path of this Model
- initialize the fIsUnique[] array from "stored AS_UNIQUE" (i.e. "stored false") published properties of every TSQLRecordClass

constructor Create; reintroduce; overload;

You should not use this constructor, but one of the overloaded versions, specifying the associated TSQLRecordClass

destructor Destroy; override;

Release associated memory

function ActionName(const Action): string;

Get the text conversion of a given Action, ready to be displayed

function AddTable(aTable: TSQLRecordClass; aTableIndexCreated: PInteger=nil): boolean;

Add the class if it doesn't exist yet
- return index in Tables[] if not existing yet and successfully added (in this case, aTableIndexCreated^ is set to the newly created index in Tables[])
- supplied class will be redefined as non-virtual: VirtualTableExternalRegister explicit call is to be made if table should be managed as external
- return FALSE if already present, or TRUE if was added to the internal list
function AddTableInherited(aTable: TSQLRecordClass): pointer;
    Add the class if it doesn't exist yet as itself or as inherited class
    - similar to AddTable(), but any class inheriting from the supplied type will be considered as
    sufficient
    - return the class which has been added, or was already there as inherited, so that could be used
    for further instance creation:
      fSQLAuthUserClass := Model.AddTableInherited(TSQLAuthUser);

function EventName(const Event): string;
    Get the text conversion of a given Event, ready to be displayed

function GetIDGenerator(aTable: TSQLRecordClass): TSynUniqueIdentifierGenerator;
    Returns the TSynUniqueIdentifierGenerator associated to a table, if any

function GetIsUnique(aTable: TSQLRecordClass; aFieldIndex: integer): boolean;
    Return TRUE if the specified field of this class was marked as unique
    - an unique field is defined as "stored AS_UNIQUE" (i.e. "stored false") in its property definition
    - reflects the internal private fIsUnique propery

function GetSQLAddField(aTableIndex, aFieldIndex: integer): RawUTF8;
    Return the UTF-8 encoded SQL source to add the corresponding field via a "ALTER TABLE"
    statement

function GetSQLCreate(aTableIndex: integer): RawUTF8;
    Return the UTF-8 encoded SQL source to create the table

function GetTableIndex(const SQLTableName: RawUTF8): integer; overload;
    Get the index of a table in Tables[]
    - expects SQLTableName to be SQL-like formatted (i.e. without TSQL[Record])

function GetTableIndex(aTable: TSQLRecordClass): integer; overload;
    Get the index of aTable in Tables[]
    - returns -1 if the table is not in the model

function GetTableIndexesFromSQLSelect(const SQL: RawUTF8): TIntegerDynArray;
    Try to retrieve one or several table index from a SQL statement
    - naive search of '... FROM Table1,Table2' pattern in the supplied SQL, using
      GetTableNamesFromSQLSelect() function

function GetTableIndexExisting(aTable: TSQLRecordClass): integer;
    Get the index of aTable in Tables[]
    - raise an EModelException if the table is not in the model

function GetTableIndexFromSQLSelect(const SQL: RawUTF8; EnsureUniqueTableInFrom: boolean): integer;
    Try to retrieve a table index from a SQL statement
    - naive search of '... FROM TableName' pattern in the supplied SQL, using
      GetTableNameFromSQLSelect() function
    - if EnsureUniqueTableInFrom is TRUE, it will check that only one Table is in the FROM clause,
      otherwise it will return the first Table specified
function GetTableIndexInheritsFrom(aTable: TSQLRecordClass): integer;
   Get the index of any class inherithing from aTable in Tables[]
   - returns -1 if no table is matching in the model

function GetTableIndexPtr(SQLTableName: PUTF8Char): integer;
   Get the index of a table in Tables[]
   - expects SQLTableName to be SQL-like formatted (i.e. without TSQL[Record])

function GetTableInherited(aTable: TSQLRecordClass): TSQLRecordClass;
   Return any class inheriting from the given table in the model
   - if the model does not contain such table, supplied aTable is returned

function GetTablesFromSQLSelect(const SQL: RawUTF8): TSQLRecordClassDynArray;
   Try to retrieve one or several TSQLRecordClass from a SQL statement
   - naive search of '... FROM Table1,Table2' pattern in the supplied SQL, using
     GetTableNamesFromSQLSelect() function

function isLocked(aRec: TSQLRecord): boolean; overload;
   Return true if a specified record is locked

function isLocked(aTable: TSQLRecordClass; aID: TID): boolean; overload;
   Return true if a specified record is locked

function Lock(aTable: TSQLRecordClass; aID: TID): boolean; overload;
   Lock a record
   - returns true on success, false if was already locked

function Lock(aRec: TSQLRecord): boolean; overload;
   Lock a record
   - returns true on success, false if was already locked

function Lock(aTableIndex: integer; aID: TID): boolean; overload;
   Lock a record
   - returns true on success, false if was already locked

function NewRecord(const SQLTableName: RawUTF8): TSQLRecord;
   Create a New TSQLRecord instance for a specific Table
   - expects SQLTableName to be SQL-like formatted (i.e. without TSQL[Record])
   - use this to create a working copy of a table's record, e.g.
   - don't forget to Free it when not used any more (use a try...finally block)
   - it's preferred in practice to directly call TSQLRecord*.Create() in your code

function RecordReference(Table: TSQLRecordClass; ID: TID): TRecordReference;
   Return the TRecordReference pointing to the specified record

function RecordReferenceTable(const Ref: TRecordReference): TSQLRecordClass;
   Return the table class correspondig to a TRecordReference

function SafeRoot: RawUTF8;
   Returns the Root property, or " if the instance is nil
function SetIDGenerator(aTable: TSQLRecordClass; aIdentifier: TSynUniqueIdentifierProcess; const aSharedObfuscationKey: RawUTF8=''): TSynUniqueIdentifierGenerator;

**Force a given table to use a TSynUniqueIdentifierGenerator for its IDs**
- will initialize a generator for the supplied table, using the given 16-bit process identifier
- you can supply an obfuscation key, which should be shared for the whole system, so that you may use FromObfuscated/ToObfuscated methods

function SQLFromSelectWhere(const Tables: array of TSQLRecordClass; const SQLSelect, SQLWhere: RawUTF8): RawUTF8;

**Compute the SQL statement to be executed for a specific SELECT on Tables**
- you can set multiple Table class in Tables: the statement will contain the table name ('SELECT T1.F1,T1.F2,T1.F3,T2.F1,T2.F2 FROM T1,T2 WHERE ..' e.g.)

function UnLock(aTableIndex: integer; aID: TID): boolean; overload;

**Unlock a specified record**
- returns true on success, false if was not already locked

function UnLock(aTable: TSQLRecordClass; aID: TID): boolean; overload;

**Unlock a specified record**
- returns true on success, false if was not already locked

function UnLock(aRec: TSQLRecord): boolean; overload;

**Unlock a specified record**
- returns true on success, false if was not already locked

function URIMatch(const URI: RawUTF8): TSQLRestModelMatch;

**Check if the supplied URI matches the model's Root property**
- allows sub-domains, e.g. if Root='root/sub1', then '/root/sub1/toto' and '/root/sub1?n=1' will match, whereas '/root/sub1nope/toto' won't
- the returned enumerates allow to check if the match was exact (e.g. 'root/sub' matches exactly Root='root'), or with character case approximation (e.g. 'Root/sub' approximates Root='root')

function VirtualTableModule(aClass: TSQLRecordClass): TSQLVirtualTableClass;

**Retrieve a Virtual Table module associated to a class**

function VirtualTableRegister(aClass: TSQLRecordClass; aModule: TSQLVirtualTableClass; const aExternalTableName: RawUTF8=''; aExternalDataBase: TObject=nil; aMappingOptions: TSQLRecordPropertiesMappingOptions=[]): boolean;

**Register a Virtual Table module for a specified class**
- to be called server-side only (Client don't need to know the virtual table implementation details, and it will increase the code size)
- aClass parameter could be either a TSQLRecordVirtual class, either a TSQLRecord class which has its kind set to rCustomForcedID or rCustomAutoID (e.g. TSQLRecordMany calling VirtualTableExternalRegister)
- optional aExternalTableName, aExternalDataBase and aMappingOptions can be used to specify e.g. connection parameters as expected by mORMotDB
- call it before TSQLRestServer.Create()

procedure PurgeOlderThan(MinutesFromNow: cardinal=30);

**Delete all the locked IDs entries, after a specified time**
- to be used to release locked records if the client crashed
- default value is 30 minutes, which seems correct for common usage
procedure SetActions(aActions: PTypeInfo);

Assign an enumeration type to the possible actions to be performed with this model
- call with the TypeInfo() pointer result of an enumeration type
- actions are handled by TSQlRecordForList in the mORMotToolBar.pas unit

procedure SetCustomCollationForAll(aFieldType: TSQLFieldType; const aCollationName: RawUTF8);

Set a custom SQlite3 text column collation for all fields of a given type for all TSQLRecord of this model
- can be used e.g. to override ALL default COLLATE SYSTEMNOCASE of RawUTF8, or COLLATE ISO8601 for TDateTime, and let the generated SQLite3 file be available outside the scope of mORMot's SQLite3 engine
- collations defined within our SynSQLite3 unit are named BINARY, NOCASE, RTRIM and our custom SYSTEMNOCASE, ISO8601, WIN32CASE, WIN32NOCASE: if you want to use the slow but Unicode ready Windows API, set for each model:
  SetCustomCollationForAll(sftUTF8Text, 'WIN32CASE');
- shall be set on both Client and Server sides, otherwise some issues may occur

procedure SetEvents(aEvents: PTypeInfo);

Assign an enumeration type to the possible events to be triggered with this class model
- call with the TypeInfo() pointer result of an enumeration type

procedure SetMaxLengthFilterForAllTextFields(IndexIsUTF8Length: boolean=false);

Allow to filter the length of all text published properties of all tables of this model
- the "index" attribute of the RawUTF8/string published properties could be used to specify a maximum length for external VARCHAR() columns
- SQLite3 will just ignore this "index" information, but it could be handy to be able to filter the value length before sending to the DB
- this method will create TSynFilterTruncate corresponding to the maximum field size specified by the "index" attribute, to validate before write
- will expect the "index" value to be in UTF-16 codepoints, unless IndexIsUTF8Length is set to TRUE, indicating UTF-8 length

procedure SetMaxLengthValidatorForAllTextFields(IndexIsUTF8Length: boolean=false);

Allow to validate length of all text published properties of all tables of this model
- the "index" attribute of the RawUTF8/string published properties could be used to specify a maximum length for external VARCHAR() columns
- SQLite3 will just ignore this "index" information, but it could be handy to be able to validate the value length before sending to the DB
- this method will create TSynValidateText corresponding to the maximum field size specified by the "index" attribute, to validate before write
- will expect the "index" value to be in UTF-16 codepoints, unless IndexIsUTF8Length is set to TRUE, indicating UTF-8 length

procedure SetVariantFieldsDocVariantOptions(const Options: TDocVariantOptions);

Customize the TDocVariant options for all variant published properties
- will change the TSQLPropInfoRTTIVariant.DocVariantOptions value
- use e.g. as SetVariantFieldDocVariantOptions(JSON_OPTIONS_FAST_EXTENDED)
- see also TSQLRecordNoCaseExtended root class
procedure UnLockAll;
Unlock all previously locked records

property Actions: PEnumType read fActions;
Performed with this model
- Actions are e.g. linked to some buttons in the User Interface

property Events: PEnumType read fEvents;
Get the enumerate type information about the possible Events to be performed with this model
- Events can be linked to actions and custom status, to provide a centralized handling of logging (e.g. in an Audit Trail table)

property Locks: TSQLLocksDynArray read fLocks;
For every table, contains a locked record list
- very fast, thanks to the use one TSQLLocks entry by table

property OnClientIdle: TOnIdleSynBackgroundThread read fOnClientIdle write fOnClientIdle;
Set a callback event to be executed in loop during client remote blocking process, e.g. to refresh the UI during a somewhat long request
- will be passed to TSQLRestClientURI.OnIdle property by TSQLRestClientURI.RegisteredClassCreateFrom() method, if applying

property Owner: TSQLRest read fRestOwner write fRestOwner;
This property value is used to auto free the database Model class
- set this property after Owner.Create() in order to have Owner.Destroy autofreeing it

property Props[aClass: TSQLRecordClass]: TSQLModelRecordProperties read GetTableProps;
The associated ORM information for a given TSQLRecord class
- raise an EModelException if aClass is not declared within this model
- returns the corresponding TableProps[] item if the class is known

property RecordReferences: TSQLModelRecordReferenceDynArray read fRecordReferences;
This array contain all TRecordReference and TSQLRecord properties
- used in TSQLRestServer.Delete() to enforce relational database coherency after deletion of a record: all other records pointing to it will be reset to 0 or deleted (if CascadeDelete is true)

property Root: RawUTF8 read fRoot write SetRoot;
The Root URI path of this Database Model
- this textual value will be used directly to compute the URI for REST routing, so it should contain only URI-friendly characters, i.e. only alphanumerical characters, excluding e.g. space or '+', otherwise an EModelException is raised

property Table[const SQLTableName: RawUTF8]: TSQLRecordClass read GetTable;
Get a class from a table name
- expects SQLTableName to be SQL-like formated (i.e. without TSQL[Record])

property TableExact[const TableName: RawUTF8]: TSQLRecordClass read GetTableExactClass;
Get a class from a table TableName (don’t truncate TSQLRecord* if necessary)
property TableProps: TSQLModelRecordPropertiesObjArray read fTableProps;
    
    The associated ORM information about all handled TSQLRecord class properties
    - this TableProps[] array will map the Tables[] array, and will allow fast direct access to the
      Tables[].RecordProps values

property Tables: TSQLRecordClassDynArray read fTables;
    
    Get the classes list (TSQLRecord descendent) of all available tables

property TablesMax: integer read fTablesMax;
    
    The maximum index of TableProps[] class properties array

property URI[aClass: TSQLRecordClass]: RawUTF8 read getURI;
    
    Get the URI for a class in this Model, as 'ModelRoot/SQLTableName'

RecordRef = object(TObject)
    
    Useful object to type cast TRecordReference type value into explicit TSQLRecordClass and ID
    - use RecordRef(Reference).TableIndex/Table/ID/Text methods to retrieve the details of a
      TRecordReference encoded value
    - use TSQLRest.Retrieve(Reference) to get a record content from DB
    - instead of From(Reference).From(), you could use the more explicit
      TSQLRecord.RecordReference(Model) or TSQLModel.RecordReference() methods or
      RecordReference() function to encode the value
    - don't change associated TSQLModel tables order, since TRecordReference depends on it to store
      the Table type
    - since 6 bits are used for the table index, the corresponding table MUST appear in the first 64
      items of the associated TSQLModel.Tables[]

Value: TID;
    
    The value itself
    - (value and 63) is the TableIndex in the current database Model
    - (value shr 6) is the ID of the record in this table
    - value=0 means no reference stored
    - we use this coding and not the opposite (Table in MSB) to minimize integer values; but special
      UTF8CompareRecord() function has to be used for sorting
    - type definition matches TRecordReference (i.e. Int64/TID) to allow typecast as such:
      aClass := PRecordRef(@Reference)^.Table(Model);

function ID: TID;
    
    Return the ID of the content

function Table(Model: TSQLModel): TSQLRecordClass;
    
    Return the class of the content in a specified TSQLModel

function TableIndex: integer;
    
    Return the index of the content Table in the TSQLModel

function Text(Model: TSQLModel): RawUTF8; overload;
    
    Get a ready to be displayed text from the stored Table and ID
    - display 'Record 2301' e.g.
function Text(Rest: TSQLRest): RawUTF8; overload;

Get a ready to be displayed text from the stored Table and ID
- display 'Record "RecordName"' e.g.

procedure From(Model: TSQLModel; aTable: TSQLRecordClass; aID: TID);

Fill Value with the corresponding parameters
- since 6 bits are used for the table index, aTable MUST appear in the first 64 items of the associated TSQLModel.Tables[] array

TSQLRecordRTreeAbstract = class(TSQLRecordVirtual)

An abstract base class, corresponding to an R-Tree table of values
- do not use this class, but either TSQLRecordRTree or TSQLRecordRTreeInteger
- an R-Tree is a special index that is designed for doing range queries. R-Trees are most commonly used in geospatial systems where each entry is a rectangle with minimum and maximum X and Y coordinates. Given a query rectangle, an R-Tree is able to quickly find all entries that are contained within the query rectangle or which overlap the query rectangle. This idea is easily extended to three dimensions for use in CAD systems. R-Trees also find use in time-domain range look-ups. For example, suppose a database records the starting and ending times for a large number of events. A R-Tree is able to quickly find all events, for example, that were active at any time during a given time interval, or all events that started during a particular time interval, or all events that both started and ended within a given time interval. And so forth. See http://www.sqlite.org/rtree.html
- any record which inherits from this class as TSQLRecordRTree must have only sftFloat (double) fields (or integer fields for TSQLRecordRTreeInteger) grouped by pairs, each as minimum- and maximum-value, up to 5 dimensions (i.e. 11 columns, including the ID property)
- since SQLite version 3.24.0 (2018-06-04), R-Tree tables can have auxiliary columns that store arbitrary data: such fields should appear after the boundary columns, and have their property name starting with '_' in the class definition; in both SQL and Where clause, the '_' will be trimmed
- note that you should better use the SynSQLite3Static unit, since an external SQLite3 .dll/.so library as supplied by the system may be outdated
- internally, the SQLite3 R-Tree extension will be implemented as a virtual table, storing coordinates/values as 32-bit floating point (single - as TSQLRecordRTree kind of ORM classes) or 32-bit integers (as TSQLRecordRTreeInteger), but will make all R-Tree computation using 64-bit floating point (double)
- as with any virtual table, the ID: TID property must be set before adding a TSQLRecordRTree to the database, e.g. to link a R-Tree representation to a regular TSQLRecord table
- queries against the ID or the coordinate ranges are almost immediate: so you can e.g. extract some coordinates box from the regular TSQLRecord table, then use a TSQLRecordRTree joined query to make the process faster; this is exactly what the TSQLRestClient.RTreeMatch method offers - of course Auxiliary Columns could avoid to make the JOIN and call RTreeMatch

class function ContainedIn(const BlobA,BlobB): boolean; virtual; abstract;

Override this class function to implement a custom SQL *_in() function
- in practice, an R-Tree index does not normally provide the exact answer but merely reduces the set of potential answers from millions to dozens: this method will be called from the *_in() SQL function to actually return exact matches
- by default, the BLOB array will be decoded via the BlobToCoord class procedure, and will create a SQL function from the class name
- used e.g. by the TSQLRestClient.RTreeMatch method
class function RTreeSQLFunctionName: RawUTF8; virtual;
Will return 'MapBox_in' e.g. for TSQLRecordMapBox

TSQLRecordRTree = class(TSQLRecordRTreeAbstract)
A base record, corresponding to an R-Tree table of floating-point values
- for instance, the following class will define a 2 dimensional RTree of floating point coordinates,
and an associated MapBox_in() function:
TSQLRecordMapBox = class(TSQLRecordRTree)
protected
  fMinX, fMaxX, fMinY, fMaxY: double;
published
  property MinX: double read fMinX write fMinX;
  property MaxX: double read fMaxX write fMaxX;
  property MinY: double read fMinY write fMinY;
  property MaxY: double read fMaxY write fMaxY;
end;
- since SQLite version 3.24.0, TSQLRecordRTree can have auxiliary columns that store arbitrary
data, having their property name starting with '_' (only in this class definition: SQL and Where
clauses will trim it)

class functioncontainedIn(const BlobA,BlobB): boolean; override;
Override this class function to implement a custom SQL *_in() function
- by default, the BLOB array will be decoded via the BlobToCoord() class procedure, and will
create a SQL function from the class name
- used e.g. by the TSQLRestClient.RTreeMatch method

class procedure BlobToCoord(const InBlob; var OutCoord: TSQLRecordTreeCoords); virtual;
Override this class function to implement a custom box coordinates from a given BLOB content
- by default, the BLOB array will contain a simple array of double
- but you can override this method to handle a custom BLOB field content, intended to hold
some kind of binary representation of the precise boundaries of the object, and convert it into
box coordinates as understood by the ContainedIn() class function
- the number of pairs in OutCoord will be taken from the current number of published double
properties
- used e.g. by the TSQLRest.RTreeMatch method
TSQLRecordRTreeInteger = class(TSQLRecordRTreeAbstract)

A base record, corresponding to an R-Tree table of 32-bit integer values
- for instance, the following class will define a 2 dimensional RTree of 32-bit integer coordinates,
and an associated MapBox_in() function:

TSQLRecordMapBox = class(TSQLRecordRTree)
protected
  fMinX, fMaxX, fMinY, fMaxY: integer;
published
  property MinX: integer read fMinX write fMinX;
  property MaxX: integer read fMaxX write fMaxX;
  property MinY: integer read fMinY write fMinY;
  property MaxY: integer read fMaxY write fMaxY;
end;

- since SQLite version 3.24.0, TSQLRecordRTreeInteger can have auxiliary columns that store arbitrary data, having their property name starting with '_' (only in this class definition: SQL and Where clauses will trim it)

class function ContainedIn(const BlobA,BlobB): boolean; override;
  Override this class function to implement a custom SQL *_in() function
  - by default, the BLOB array will be decoded via the BlobToCoord() class procedure, and will create a SQL function from the class name
  - used e.g. by the TSQLRest.RTreeMatch method

class procedure BlobToCoord(const InBlob; var OutCoord: TSQLRecordTreeCoordsInteger); virtual;
  Override this class function to implement a custom box coordinates from a given BLOB content
  - by default, the BLOB array will contain a simple array of integer
  - but you can override this method to handle a custom BLOB field content, intended to hold some kind of binary representation of the precise boundaries of the object, and convert it into box coordinates as understood by the ContainedIn() class function
  - the number of pairs in OutCoord will be taken from the current number of published integer properties
  - used e.g. by the TSQLRest.RTreeMatch method
TSQLRecordFTS3 = class(TSQLRecordVirtual)

A base record, corresponding to a FTS3 table, i.e. implementing full-text
- FTS3/FTS4/FTS5 tables are SQLite virtual tables allowing users to perform full-text searches on a
set of documents. The most common (and effective) way to describe full-text searches is "what
Google, Yahoo and Altavista do with documents placed on the World Wide Web". Users input a
term, or series of terms, perhaps connected by a binary operator or grouped together into a
phrase, and the full-text query system finds the set of documents that best matches those terms
considering the operators and groupings the user has specified. See http://sqlite.org/fts3.html
- any record which inherits from this class must have only sftUTF8Text (RawUTF8) fields - with
Delphi 2009+, you can have string fields
- this record has its fID: TID property which may be published as DocID, to be consistent with
SQLite3 praxis, and reflect that it points to an ID of another associated TSQLRecord
- a good approach is to store your data in a regular TSQLRecord table, then store your text content
in a separated FTS3 table, associated to this TSQLRecordFTS3 table via its ID/DocID
- the ID/DocID property can be set when the record is added, to retrieve any associated
TSQLRecord (note that for a TSQLRecord record, the ID property can't be set at adding, but is
calculated by the engine)
- static tables don't handle TSQLRecordFTS3 classes
- by default, the FTS3 engine ignore all characters >= #80, but handle low-level case insensitivity (i.e.
'A'..'Z') so you must keep your points with the same range for upper case
- by default, the "simple" tokenizer is used, but you can inherits from TSQLRecordFTS3Porter class
if you want a better English matching, using the Porter Stemming algorithm, or
TSQLRecordFTS3Unicode61 for Unicode support - see http://sqlite.org/fts3.html#tokenizer
- you can select either the FTS3 engine, or the more efficient (and new) FTS4 engine (available
since version 3.7.4), by using the TSQLRecordFTS4 type, or TSQLRecordFTS5 for the latest (and
preferred) FTS5 engine
- in order to make FTS queries, use the dedicated TSQLRest.FTSMatch method, with the MATCH
operator (you can use regular queries, but you must specify 'RowID' instead of 'DocID' or 'ID'
because of FTS3 virtual table specificity):

```delphi
var IDs: TIDDynArray;
if FTSMatch(TSQLMyFTS3Table,'text MATCH "linu*"',IDs) then
// you have all matching IDs in IDs[]
```

- by convention, inherited class name could specify a custom stemming algorithm by starting with
"TSQLRecordFTS3", and adding the algorithm name as suffix, e.g. TSQLRecordFTS3Porter will
create a "tokenize=porter" virtual table

class function OptimizeFTS3Index(Server: TSQLRestServer): boolean;

Optimize the FTS3 virtual table
- this causes FTS3 to merge all existing index b-trees into a single large b-tree containing the
entire index. This can be an expensive operation, but may speed up future queries. See
http://sqlite.org/fts3.html#section_1_2
- this method must be called server-side
- returns TRUE on success
property DocID: TID read GetID write fID;

This DocID property map the internal Row_ID property
- but you can set a value to this property before calling the Add() method, to associate this
TSQLRecordFTS3 to another TSQLRecord
- ID property is read-only, but this DocID property can be written/set
- internally, we use RowID in the SQL statements, which is compatible with both TSQLRecord and
TSQLRecordFTS3 kind of table

TSQLRecordFTS3Porter = class(TSQLRecordFTS3)

This base class will create a FTS3 table using the Porter Stemming algorithm
- see http://sqlite.org/fts3.html#tokenizer
- will generate tokenize=porter by convention from the class name

TSQLRecordFTS3Unicode61 = class(TSQLRecordFTS3)

This base class will create a FTS3 table using the Unicode61 Stemming algorithm
- see http://sqlite.org/fts3.html#tokenizer
- will generate tokenize=unicode64 by convention from the class name

TSQLRecordFTS4 = class(TSQLRecordFTS3)

A base record, corresponding to a FTS4 table, which is an enhancement to FTS3
- FTS3 and FTS4 are nearly identical. They share most of their code in common, and their
interfaces are the same. The only difference is that FTS4 stores some additional information about
the document collection in two of new FTS shadow tables. This additional information allows FTS4
to use certain query performance optimizations that FTS3 cannot use. And the added information
permits some additional useful output options in the matchinfo() function.
- for newer applications, TSQLRecordFTS5 is recommended; though if minimal disk usage or
compatibility with older versions of SQLite are important, then TSQLRecordFTS3 will usually serve
just as well
- see http://sqlite.org/fts3.html#section_1_1
- by convention, inherited class name could specify a custom stemming algorithm by starting with
"TSQLRecordFTS4", and adding the algorithm name as suffix, e.g. TSQLRecordFTS'Porter will
create a "tokenize=porter" virtual table

class procedure InitializeTable(Server: TSQLRestServer; const FieldName: RawUTF8;
Options: TSQLInitializeTableOptions); override;

This overridden method will create TRIGGERS for FTSWithoutContent()

TSQLRecordFTS4Porter = class(TSQLRecordFTS4)

This base class will create a FTS4 table using the Porter Stemming algorithm
- see http://sqlite.org/fts3.html#tokenizer
- will generate tokenize=porter by convention from the class name

TSQLRecordFTS4Unicode61 = class(TSQLRecordFTS4)

This base class will create a FTS4 table using the Unicode61 Stemming algorithm
- see http://sqlite.org/fts3.html#tokenizer
- will generate tokenize=unicode64 by convention from the class name
TSQLRecordFTS5 = class(TSQLRecordFTS4)

A base record, corresponding to a FTS5 table, which is an enhancement to FTS4
- FTS5 is a new version of FTS4 that includes various fixes and solutions for problems that could not be fixed in FTS4 without sacrificing backwards compatibility
- for newer applications, TSQLRecordFTS5 is recommended; though if minimal disk usage or compatibility with older versions of SQLite are important, then TSQLRecordFTS3/TSQLRecordFTS4 will usually serve just as well
- see https://sqlite.org/fts5.html#appendix_a
- by convention, inherited class name could specify a custom stemming algorithm by starting with "TSQLRecordFTS5", and adding the algorithm name as suffix, e.g. TSQLRecordFTS5Porter will create a "tokenize=porter" virtual table

TSQLRecordFTS5Porter = class(TSQLRecordFTS5)

This base class will create a FTS5 table using the Porter Stemming algorithm
- see https://sqlite.org/fts5.html#tokenizers
- will generate tokenize=porter by convention from the class name

TSQLRecordFTS5Unicode61 = class(TSQLRecordFTS5)

This base class will create a FTS5 table using the Unicode61 Stemming algorithm
- see https://sqlite.org/fts5.html#tokenizers
- will generate tokenize=unicode64 by convention from the class name
TSQLEmany = class(TSQLRecord)

Handle "has many" and "has many through" relationships
- many-to-many relationship is tracked using a table specifically for that relationship, turning the relationship into two one-to-many relationships pointing in opposite directions
- by default, only two TSQLRecord (i.e. INTEGER) fields must be created, named "Source" and "Dest", the first pointing to the source record (the one with a TSQLRecordMany published property) and the second to the destination record
- you should first create a type inheriting from TSQLRecordMany, which will define the pivot table, providing optional "through" parameters if needed

TSQLDest = class(TSQLRecord);
TSQLSource = class;
TSQLDestPivot = class(TSQLRecordMany)
private
  fSource: TSQLSource;
  fDest: TSQLDest;
  fTime: TDateTime;
published
  property Source: TSQLSource read fSource; // map Source column
  property Dest: TSQLDest read fDest; // map Dest column
  property AssociationTime: TDateTime read fTime write fTime;

TSQLSource = class(TSQLRecord)
private
  fDestList: TSQLDestPivot;
published
  DestList: TSQLDestPivot read fDestList;

- in all cases, at least two 'Source' and 'Dest' published properties must be declared as TSQLRecord children in any TSQLRecordMany descendant because they will always be needed for the 'many to many' relationship
- when a TSQLRecordMany published property exists in a TSQLRecord, it is initialized automatically by TSQLRecord.Create
- to add some associations to the pivot table, use the ManyAdd() method
- to retrieve an association, use the ManySelect() method
- to delete an association, use the ManyDelete() method
- to read all Dest records IDs, use the DestGet() method
- to read the Dest records and the associated "through" fields content, use FillMany then FillRow, FillOne and FillRewind methods to loop through records
- to read all Source records and the associated "through" fields content, FillManyFromDest then FillRow, FillOne and FillRewind methods
- to read all Dest IDs after a join to the pivot table, use DestGetJoined

constructor Create; override;
  Initialize this instance, and needed internal fields
  - will set protected fSourceID/fDestID fields

function DestGet(aClient: TSQLRest; aSourceID: TID; out DestIDs: TIDDynArray): boolean; overload;

Retrieves all Dest items IDs associated to the specified Source
function DestGet(aclient: TSQLRest; out DestIDs: TIDDynArray): boolean; overload;

Retrieve all Dest items IDs associated to the current Source ID
- source ID is taken from the fSourceID field (set by TSQLRecord.Create)
- note that if the Source record has just been added, fSourceID is not set, so this method will fail:
please call the other overloaded method

function DestGetJoined(aclient: TSQLRest; const aDestWhereSQL: RawUTF8; aSourceID: TID): TSQLRecord; overload;

Create a Dest record, then FillPrepare() it to retrieve all Dest items associated to the current or specified Source ID, adding a WHERE condition against the Dest rows
- if aSourceID is 0, the value is taken from current fSourceID field (set by TSQLRecord.Create)
- aDestWhereSQL can specify the Dest table name in the statement, e.g. 'Salary>:(1000): AND Salary<:(2000):' according to TSQLRecordMany properties - note that you should better use such inlined parameters as
  FormatUTF8('Salary>? AND Salary<?',[],[1000,2000])

function DestGetJoined(aclient: TSQLRest; const aDestWhereSQL: RawUTF8; aSourceID: TID; out DestIDs: TIDDynArray): boolean; overload;

Retrieve all Dest items IDs associated to the current or specified Source ID, adding a WHERE condition against the Dest rows
- if aSourceID is 0, the value is taken from current fSourceID field (set by TSQLRecord.Create)
- aDestWhereSQL can specify the Dest table name in the statement, e.g. 'Salary>:(1000): AND Salary<:(2000):' - note that you should better use inlined parameters for faster processing on server, so you may use the more convenient function
  FormatUTF8('Salary>? AND Salary<?',[],[1000,2000])
- this is faster than a manual FillMany() then loading each Dest, because the condition is executed in the SQL statement by the server

function DestGetJoinedTable(aclient: TSQLRest; const aDestWhereSQL: RawUTF8; aSourceID: TID; JoinKind: TSQLRecordManyJoinKind; const aCustomFieldsCSV: RawUTF8=''): TSQLTable;

Create a TSQLTable, containing all specified Fields, after a JOIN associated to the current or specified Source ID
- the Table will have the fields specified by the JoinKind parameter
- aCustomFieldsCSV can be used to specify which fields must be retrieved (for jkDestFields, jkPivotFields, jkPivotAndDestFields) - default is all
- if aSourceID is 0, the value is taken from current fSourceID field (set by TSQLRecord.Create)
- aDestWhereSQL can specify the Dest table name in the statement, e.g. 'Salary>:(1000): AND Salary<:(2000):' according to TSQLRecordMany properties - note that you should better use such inlined parameters as
  FormatUTF8('Salary>? AND Salary<?',[],[1000,2000])
function FillMany(aClient: TSQLRest; aSourceID: TID=0; const aAndWhereSQL: RawUTF8=''): integer;

Retrieve all records associated to a particular source record, which has a TSQLRecordMany property
- returns the Count of records corresponding to this aSource record
- the records are stored in an internal TSQLTable, refered in the private fTable field, and initialized via a FillPrepare call: all Dest items are therefore accessible with standard FillRow, FillOne and FillRewind methods
- use a "for .." loop or a "while FillOne do ..." loop to iterate through all Dest items, getting also any additional 'through' columns
- if source ID parameter is 0, the ID is taken from the fSourceID field (set by TSQLRecord.Create)
- note that if the Source record has just been added, fSourceID is not set, so this method will fail: please specify aSourceID parameter with the one just added/created
- the optional aAndWhereSQL parameter can be used to add any additional condition to the WHERE statement (e.g. 'Salary>:1000: AND Salary<:2000:') according to TSQLRecordMany properties - note that you should better use inlined parameters for faster processing on server, so you may call e.g.

```
aRec.FillMany(Client,0,FormatUTF8('Salary>:100 AND Salary<:',[],[1000,2000]));
```

function FillManyFromDest(aClient: TSQLRest; aDestID: TID; const aAndWhereSQL: RawUTF8=''): integer;

Retrieve all records associated to a particular Dest record, which has a TSQLRecordMany property
- returns the Count of records corresponding to this aSource record
- use a "for .." loop or a "while FillOne do ..." loop to iterate through all Dest items, getting also any additional 'through' columns
- the optional aAndWhereSQL parameter can be used to add any additional condition to the WHERE statement (e.g. 'Salary>:1000: AND Salary<:2000:') according to TSQLRecordMany properties - note that you should better use inlined parameters for faster processing on server, so you may call e.g.

```
aRec.FillManyFromDest(Client,DestID,FormatUTF8('Salary>:100 AND Salary<:',[],[1000,2000]));
```

function IDWhereSQL(aClient: TSQLRest; aID: TID; isDest: boolean; const aAndWhereSQL: RawUTF8=''): RawUTF8;

Get the SQL WHERE statement to be used to retrieve the associated records according to a specified ID
- search for aID as Source ID if isDest is FALSE
- search for aID as Dest ID if isDest is TRUE
- the optional aAndWhereSQL parameter can be used to add any additional condition to the WHERE statement (e.g. 'Salary>:1000: AND Salary<:2000:') according to TSQLRecordMany properties - note that you should better use such inlined parameters e.g. calling

```
FormatUTF8('Salary>: AND Salary<:',[],[1000,2000])
```
function ManyAdd(aClient: TSQLRest; aDestID: TID; NoDuplicates: boolean=false): boolean; overload;
    Add a Dest record to the current Source record list
    - source ID is taken from the fSourceID field (set by TSQLRecord.Create)
    - note that if the Source record has just been added, fSourceID is not set, so this method will fail:
      please call the other overloaded method

function ManyAdd(aClient: TSQLRest; aSourceID, aDestID: TID; NoDuplicates: boolean=false; aUseBatch: TSQLRestBatch=nil): boolean; overload;
    Add a Dest record to the Source record list
    - returns TRUE on success, FALSE on error
    - if NoDuplicates is TRUE, the existence of this Source/Dest ID pair is first checked
    - current Source and Dest properties are filled with the corresponding TRecordReference values
      corresponding to the supplied IDs
    - any current value of the additional fields are used to populate the newly created content (i.e.
      all published properties of this record)
    - if aUseBatch is set, it will use this TSQLRestBatch.Add() instead of the slower aClient.Add() method

function ManyDelete(aClient: TSQLRest; aDestID: TID): boolean; overload;
    Will delete the record associated with the current source and a specified Dest
    - source ID is taken from the fSourceID field (set by TSQLRecord.Create)
    - note that if the Source record has just been added, fSourceID is not set, so this method will fail:
      please call the other overloaded method

function ManyDelete(aClient: TSQLRest; aSourceID, aDestID: TID; aUseBatch: TSQLRestBatch=nil): boolean; overload;
    Will delete the record associated with a particular Source/Dest pair
    - will return TRUE if the pair was found and successfully deleted
    - if aUseBatch is set, it will use this TSQLRestBatch.Delete() instead of the slower aClient.Delete() method

function ManySelect(aClient: TSQLRest; aDestID: TID): boolean; overload;
    Will retrieve the record associated with the current source and a specified Dest
    - source ID is taken from the fSourceID field (set by TSQLRecord.Create)
    - note that if the Source record has just been added, fSourceID is not set, so this method will fail:
      please call the other overloaded method

function ManySelect(aClient: TSQLRest; aSourceID, aDestID: TID): boolean; overload;
    Will retrieve the record associated with a particular Source/Dest pair
    - will return TRUE if the pair was found
    - in this case, all "through" columns are available in the TSQLRecordMany field instance

function SourceGet(aClient: TSQLRest; aDestID: TID; out SourceIDs: TIDDynArray): boolean;
    Retrieve all Source items IDs associated to the specified Dest ID

TSQLRecordLog = class(TSQLRecord)
    A base record, with a JSON-logging capability
    - used to store a log of events into a JSON text, easy to be displayed with a TSQLTableToGrid
    - this log can then be stored as a RawUTF8 field property into a result record, for instance
**constructor** CreateFrom(OneLog: TSQLRecord; const aJSON: RawUTF8);

*Initialize the internal storage with a supplied JSON content*
- this JSON content must follow the format retrieved by LogTableJSON and LogTableJSONFrom methods

**destructor** Destroy; **override**;

*Release the private fLogTableWriter and fLogTableStorage objects*

**function** LogCurrentPosition: integer;

*Returns the internal position of the Log content*
- use this value to later retrieve a log range with LogTableJSONFrom()

**function** LogTableJSON: RawUTF8;

*Returns the JSON data as added by previous call to Log()*
- JSON data is in not-expanded format
- this function can be called multiple times

**function** LogTableJSONFrom(StartPosition: integer): RawUTF8;

*Returns the log JSON data from a given start position*
- StartPosition was retrieved previously with LogCurrentPosition
- if StartPosition=0, the whole Log content is returned
- multiple instances of LogCurrentPosition/LogTableJSONFrom() can be used at once

**procedure** Log(OneLog: TSQLRecord);

*Add the value of OneLog to the Log Table JSON content*
- the ID property of the supplied OneLog record is incremented before adding

**property** LogTableRowCount: integer **read** fLogTableRowCount;

*The current associated Log Table rows count value*
- is incremented every time Log() method is called
- will be never higher than MaxLogTableRowCount below (if set)

**property** MaxLogTableRowCount: integer **read** fMaxLogTableRowCount;

*If the associated Log Table rows count reaches this value, the first data row will be trimed*
- do nothing if value is left to 0 (which is the default)
- total rows count won't never be higher than this value
- used to spare memory usage

**TSQLRecordSigned** = **class**(TSQLRecord)

*Common ancestor for tables with digitally signed RawUTF8 content*
- content is signed according to a specific User Name and the digital signature date and time
- internally uses the very secure SHA-256 hashing algorithm for performing the digital signature

**function** CheckSignature(const Content: RawByteString): boolean;

*Returns true if this record content is correct according to the stored digital Signature

**function** SetAndSignContent(const UserName: RawUTF8; **const** Content: RawByteString; ForcedSignatureTime: Int64=0): boolean;

*Use this procedure to sign the supplied Content of this record for a specified UserName, with the current Date and Time*
- SHA-256 hashing is used internally
- returns true if signed successfully (not already signed)
function SignedBy: RawUTF8;
  Retrieve the UserName who digitally signed this record
  - returns "" if was not digitally signed

procedure UnSign;
  Reset the stored digital signature
  - SetAndSignContent() can be called after this method

property Signature: RawUTF8 read fSignature write fSignature;
  As the Content of this record is added to the database, its value is hashed and stored as
  'UserName/03A35C92....' into this property
  - secured SHA-256 hashing is used internally
  - digital signature is allowed only once: this property is written only once
  - this property is defined here to allow inherited to just declared the name in its published
    section:
    property Signature;

property SignatureTime: TTimeLog read fSignatureTime write fSignatureTime;
  Time and date of signature
  - if the signature is invalid, this field will contain numerical 1 value
  - this property is defined here to allow inherited to just declared the name in its published
    section:
    property SignatureTime;

TSQlRecordTimed = class(TSQLRecord)
  A base record, which will have creation and modification timestamp fields

property Created: TCreateTime read fCreated write fCreated;
  Will be filled by the ORM when this item will be created in the database

property Modified: TModTime read fModified write fModified;
  Will be filled by the ORM each time this item will be written in the database

TSQlRecordInterfaced = class(TSQLRecord)
  Common ancestor for tables which should implement any interface
  - by default, TSQLRecord does not implement any interface: this does make sense for
    performance and resource use reasons
  - inherit from this class if you want your class to implement the needed IInterface methods
    (QueryInterface/AddRef/Release)

TServiceMethodArgument = object(TObject)
  Describe a service provider method argument

ArgTypeInfo: PTypeInfo;
  The low-level RTTI information of this argument

ArgTypeName: PShortString;
  The type name, as declared in Delphi
DynArrayWrapper: TDynArray;
   A TDynArray wrapper initialized properly for this smvDynArray

FPRegisterIdent: integer;
   Used to specify if a floating-point argument is passed as register
      - contains always 0 for x86/x87
      - contains 1 for XMM0, 2 for XMM1, ..., 4 for XMM3 for x64
      - contains 1 for D0, 2 for D1, ..., 8 for D7 for armhf
      - contains 1 for V0, 2 for V1, ..., 8 for V7 for aarch64

IndexVar: integer;
   Index of the associated variable in the local array[ArgsUsedCount[]]
      - for smdConst argument, contains -1 (no need to a local var: the value will be on the stack only)

InStackOffset: integer;
   Byte offset in the CPU stack of this argument
      - may be -1 if pure register parameter with no backup on stack (x86)

ParamName: PShortString;
   The argument name, as declared in Delphi

RegisterIdent: integer;
   Used to specify if the argument is passed as register
      - contains 0 if parameter is not a register
      - contains 1 for EAX, 2 for EDX and 3 for ECX registers for x86
      - contains 1 for RCX, 2 for RDX, 3 for R8, and 4 for R9, with a backing store on the stack for x64
      - contains 1 for R0, 2 for R1 ... 4 for R3, with a backing store on the stack for arm
      - contains 1 for X0, 2 X1 ... 8 for X7, with a backing store on the stack for aarch64

SizeInBinary: integer;
   Hexadecimal binary size (in bytes) of this smv64 ordinal value
      - set only if ValueType=smvBinary

SizeInStack: integer;
   Size (in bytes) of this argument on the stack

SizeInStorage: integer;
   Size (in bytes) of this smv64 ordinal value
      - e.g. depending of the associated kind of enumeration

ValueDirection: TServiceMethodValueDirection;
   The variable direction as defined at code level

ValueKindAsm: TServiceMethodValueAsm;
   How the variable is to be passed at asm level

ValueType: TServiceMethodValueType;
   We do not handle all kind of Delphi variables

ValueVar: TServiceMethodValueVar;
   How the variable may be stored
function FromJSON(const MethodName: RawUTF8; var R: PUTF8Char; V: pointer; Error: PShortString; DVO: TDocVariantOptions): boolean;
    Unserialize a JSON value into this argument

function IsDefault(V: pointer): boolean;
    Check if the supplied argument value is the default (e.g. 0, " or null)

procedure AddAsVariant(var Dest: TDocVariantData; V: pointer);
    Add a value into a TDocVariant object or array
    - Dest should already have set its Kind to either dvObject or dvArray

procedure AddDefaultJSON(WR: TTextWriter);
    Append the default JSON value corresponding to this argument
    - includes a pending ','

procedure AddJSON(WR: TTextWriter; V: pointer; ObjectOptions: TTextWriterWriteObjectOptions=[woDontStoreDefault]);
    Append the JSON value corresponding to this argument
    - includes a pending ','

procedure AddJSONEscaped(WR: TTextWriter; V: pointer);
    Append the value corresponding to this argument as within a JSON string
    - will escape any JSON string character, and include a pending ','

procedure AddValueJSON(WR: TTextWriter; const Value: RawUTF8);
    Append the JSON value corresponding to this argument, from its text value
    - includes a pending ','

procedure AsJson(var DestValue: RawUTF8; V: pointer);
    Convert a value into its JSON representation

procedure AsVariant(var DestValue: variant; V: pointer; Options: TDocVariantOptions);
    Convert a value into its variant representation
    - complex objects will be converted into a TDocVariant, after JSON serialization: variant conversion options may e.g. be retrieve from TInterfaceFactory.DocVariantOptions

procedure FixValue(var Value: variant);
    Normalize a value containing one input or output argument
    - sets and enumerates will be translated to strings (also in embedded objects and T*ObjArray)

procedure FixValueAndAddToObject(const Value: variant; var DestDoc: TDocVariantData);
    Normalize a value containing one input or output argument, and add it to a destination variant Document
    - sets and enumerates will be translated to strings (also in embedded objects and T*ObjArray)

procedure SerializeToContract(WR: TTextWriter);
    Serialize the argument into the TServiceContainer.Contract JSON format
    - non standard types (e.g. clas, enumerate, dynamic array or record) are identified by their type identifier - so contract does not extend up to the content of such high-level structures

procedure SetFromRTTI(var P: PByte);
    Set ArgTypeName and ArgTypeInfo values from RTTI
TServiceMethod = object(TObject)

Describe an interface-based service provider method

Args: TServiceMethodArgumentDynArray;

Describe expected method arguments
- Args[0] always is smvSelf
- if method is a function, an additional smdResult argument is appended

ArgsInFirst: shortint;
The index of the first const / var argument in Args[]

ArgsInLast: shortint;
The index of the last const / var argument in Args[]

ArgsInputIsOctetStream: boolean;
True if there is a single input parameter as RawByteString/TSQLRawBlob
- TSQLRestRoutingREST.ExecuteSOAPByInterface will identify binary input with mime-type
  'application/octet-stream' as expected

ArgsInputValuesCount: byte;
The number of const / var parameters in Args[]
- i.e. the number of elements in the input JSON array

ArgsManagedFirst: shortint;
The index of the first argument expecting manual stack initialization
- set if there is any smvObject, smvDynArray, smvRecord, smvInterface or smvVariant

ArgsManagedLast: shortint;
The index of the last argument expecting manual stack initialization
- set if there is any smvObject, smvDynArray, smvRecord, smvInterface or smvVariant

ArgsNotResultLast: shortint;
The index of the last argument in Args[], excepting result

ArgsOutFirst: shortint;
The index of the first var / out / result argument in Args[]

ArgsOutLast: shortint;
The index of the last var / out / result argument in Args[]

ArgsOutNotResultLast: shortint;
The index of the last var / out argument in Args[]

ArgsOutputValuesCount: byte;
The number of var / out parameters + in Args[]
- i.e. the number of elements in the output JSON array or object

ArgsResultIndex: shortint;
The index of the result pseudo-argument in Args[]
- is -1 if the method is defined as a (not a function)
ArgsResultIsServiceCustomAnswer: boolean;

  True if the result is a TServiceCustomAnswer record
  - that is, a custom Header+Content BLOB transfer, not a JSON object

ArgsSizeInStack: cardinal;

  Needed CPU stack size (in bytes) for all arguments
  - under x64, does not include the backup space for the four registers

ArgsUsed: TServiceMethodValueTypes;

  Contains all used kind of arguments

ArgsUsedCount: array[TServiceMethodValueVar] of byte;

  Contains the count of variables for all used kind of arguments

DefaultResult: RawUTF8;

  The method default result, formatted as a JSON array
  - example of content may be '[]' for a procedure or '[0]' for a function
  - any var/out and potential function result will be set as a JSON array of values, with 0 for numerical values, '"' for textual values, false for booleans, [] for dynamic arrays, a void record serialized as expected (including customized serialization) and null for objects

ExecutionMethodIndex: byte;

  Method index in the original (non emulated) interface
  - our custom methods start at index 3 (RESERVED_VTABLE_SLOTS), since QueryInterface, _AddRef, and _Release are always defined by default
  - so it maps TServiceFactory.Interface.Methods[ExecutionMethodIndex - 3]

HasSPIParams: TServiceMethodValueDirections;

  The directions of arguments with vIsSPI defined in Args[].ValueKindAsm

HierarchyLevel: byte;

  Is 0 for the root interface, 1..n for all inherited interfaces

InterfaceDotMethodName: RawUTF8;

  The fully qualified dotted method name, including the interface name
  - as used by TServiceContainerInterfaceMethod.InterfaceDotMethodName
  - match the URI fullpath name, e.g. 'Calculator.Add'

IsInherited: boolean;

  TRUE if the method is inherited from another parent interface

URI: RawUTF8;

  The method URI, i.e. the method name
  - as declared in Delphi code, e.g. 'Add' for ICalculator.Add
  - this property value is hashed internally for faster access

function ArgIndex(ArgName: PUTF8Char; ArgNameLen: integer; Input: boolean): integer;

  Retrieve an argument index in Args[] from its name
  - search is case insensitive
  - if Input is TRUE, will search within const / var arguments
  - if Input is FALSE, will search within var / out / result arguments
  - returns -1 if not found
function ArgNext(var arg: integer; Input: boolean): boolean;

*Find the next argument index in Args[]*
- if Input is TRUE, will search within const / var arguments
- if Input is FALSE, will search within var / out / result arguments
- returns true if arg is the new value, false otherwise

function ArgsArrayToObject(P: PUTF8Char; Input: boolean): RawUTF8;

*Convert parameters encoded as a JSON array into a JSON object*
- if Input is TRUE, will handle const / var arguments
- if Input is FALSE, will handle var / out / result arguments

function ArgsCommandLineToObject(P: PUTF8Char; Input: boolean; RaiseExceptionOnUnknownParam: boolean=false): RawUTF8;

*Convert parameters encoded as name=value or name="value" or name='{somejson}' into a JSON object*
- on Windows, use double-quotes (""") anywhere you expect single-quotes (")
- as expected e.g. from a command line tool
- if Input is TRUE, will handle const / var arguments
- if Input is FALSE, will handle var / out / result arguments

function ArgsNames(Input: Boolean): TRawUTF8DynArray;

*Returns a dynamic array list of all parameter names*
- if Input is TRUE, will handle const / var arguments
- if Input is FALSE, will handle var / out / result arguments

procedure ArgsAsDocVariantFix(var ArgsObject: TDocVariantData; Input: boolean);

*Normalize a TDocVariant containing the input or output arguments values*
- "normalization" will ensure sets and enums are serialized as text
- if Input is TRUE, will handle const / var arguments
- if Input is FALSE, will handle var / out / result arguments

procedure ArgsAsDocVariantObject(const ArgsParams: TDocVariantData; var ArgsObject: TDocVariantData; Input: boolean);

*Convert a TDocVariant array containing the input or output arguments values in order, into an object with named parameters*
- here sets and enums will keep their current values, mainly numerical
- if Input is TRUE, will handle const / var arguments
- if Input is FALSE, will handle var / out / result arguments

procedure ArgsStackAsDocVariant(const Values: TPPPointerDynArray; out Dest: TDocVariantData; Input: Boolean);

*Computes a TDocVariant containing the input or output arguments values*
- Values[] should point to the input/output raw binary values, as stored in TServiceMethodExecute.Values during execution

procedure ArgsValuesAsDocVariant(Kind: TServiceMethodParamsDocVariantKind; out Dest: TDocVariantData; const Values: TVariantDynArray; Input: boolean; Options: TDocVariantOptions=[dvoReturnNullForUnknownProperty,dvoValueCopiedByReference]);

*Computes a TDocVariant containing the input or output arguments values*
- Values[] should contain the input/output raw values as variant
- Kind will specify the expected returned document layout
**TSQLRecordServiceLog** = class(TSQLRecordNoCaseExtended)  
Common ancestor for storing interface-based service execution statistics  
- each call could be logged and monitored in the database  
- TServiceMethodExecute could store all its calls in such a table  
- enabled on server side via either TServiceFactoryServer.SetServiceLog or TServiceContainerServer.SetServiceLog method

**class procedure InitializeTable(Server: TSQLRestServer; const FieldName: RawUTF8; Options: TSQLInitializeTableOptions); override;**

*Overridden method creating an index on the Method/MicroSec columns*

**property Input:** variant read fInput write fInput;

*The input parameters, as a JSON document*
- will be stored in JSON_OPTIONS_FAST_EXTENDED format, i.e. with shortened field names, for smaller TEXT storage
- content may be searched using JsonGet/JsonHas SQL functions on a SQLite3 storage, or with direct document query under MongoDB/PostgreSQL

**property IP:** RawUTF8 read fIP write fIP;

*If not localhost/127.0.0.1, the remote IP address*

**property Method:** RawUTF8 read fMethod write fMethod;

*The 'interface.method' identifier of this call*
- this column will be indexed, for fast SQL queries, with the MicroSec column (for performance tuning)

**property MicroSec:** integer read fMicroSec write fMicroSec;

*Execution time of this method, in micro seconds*

**property Output:** variant read fOutput write fOutput;

*The output parameters, as a JSON document, including result: for a function*
- will be stored in JSON_OPTIONS_FAST_EXTENDED format, i.e. with shortened field names, for smaller TEXT storage
- content may be searched using JsonGet/JsonHas SQL functions on a SQLite3 storage, or with direct document query under MongoDB/PostgreSQL

**property Session:** integer read fSession write fSession;

*The Session ID, if there is any*

**property Time:** TModTime read fTime write fTime;

*Will be filled by the ORM when this record is written in the database*

**property User:** integer read fUser write fUser;

*The User ID, if there is an identified Session*

**TSQLRecordServiceNotifications** = class(TSQLRecordServiceLog)  
Execution statistics used for DB-based asynchronous notifications  
- as used by TServiceFactoryClient.SendNotifications  
- here, the Output column may contain the information about an error occurred during process
class function LastEventsAsObjects(Rest: TSQLRest; LastKnownID: TID; Limit: integer; Service: TInterfaceFactory; out Dest: TDocVariantData; const MethodName: RawUTF8 = 'Method'; IDAsHexa: boolean = false): boolean;

Search for pending events since a supplied ID
- returns FALSE if no notification was found
- returns TRUE ad fill a TDocVariant array of JSON Objects, including "ID": field, and Method as "MethodName": field

function SaveInputAsObject(Service: TInterfaceFactory; const MethodName: RawUTF8 = 'Method'; IDAsHexa: boolean = false): variant;

Allows to convert the Input array into a proper single JSON Object
- "ID": field will be included, and Method as "MethodName": field

class procedure InitializeTable(Server: TSQLRestServer; const FieldName: RawUTF8; Options: TSQLInitializeTableOptions); override;

This overridden method will create an index on the 'Sent' column

procedure SaveFillInputsAsObjects(Service: TInterfaceFactory; out Dest: TDocVariantData; const MethodName: RawUTF8 = 'Method'; IDAsHexa: boolean = false);

Run FillOne and SaveInputAsObject into a TDocVariant array of JSON Objects
- "ID": field will be included, and Method as "MethodName": field

property Sent: TTimeLog read fSent write fSent;

When this notification has been sent
- equals 0 until it was actually notified

TServiceMethodExecute = class(TObject)
Execute a method of a TInterfacedObject instance, from/to JSON

constructor Create(aMethod: PServiceMethod);

Initialize the execution instance

destructor Destroy; override;

Finalize the execution instance

function ExecuteJson(const Instances: array of pointer; Par: PUTF8Char; Res: TTextWriter; Error: PShortString=nil; ResAsJSONObject: boolean=false): boolean;

Execute the corresponding method of weak Invokable references
- will retrieve a JSON array of parameters from Par (as [1,"par2",3])
- will append a JSON array of results in Res, or set an Error message, or a JSON object (with parameter names) in Res if ResultAsJSONObject is set
- if one Instances[] is supplied, any exception will be propagated (unless optIgnoreException is set); if more than one Instances[] is supplied, corresponding ExecutedInstancesFailed[] property will be filled with the JSON serialized exception

function ExecuteJsonCallback(Instance: pointer; const params: RawUTF8; output: PRawUTF8): boolean;

Execute the corresponding method of one weak Invokable reference
- expect no output argument, i.e. no returned data, unless output is set
- this version will identify TInterfacedObjectFake implementations, and will call directly fnInvoke() if possible, to avoid JSON marshalling
- expect params value to be without [], just like TOnFakeInstanceInvoke
function ExecuteJsonFake(Instance: pointer; params: PUTF8Char): boolean;
    Execute directly TInterfacedObjectFake.fInvoke()
    - expect params value to be with [], just like ExecuteJson

function TempTextWriter: TJJSONSerializer;
    Allow to use an instance-specific temporary TJJSONSerializer

procedure AddInterceptor(const Hook: TServiceMethodExecuteEvent);
    Allow to hook method execution
    - if optInterceptInputOutput is defined in Options, then Sender.Input/Output fields will contain
      the execution data context when Hook is called

property BackgroundExecutionThread: TSynBackgroundThreadMethod read fBackgroundExecutionThread;
    Reference to the background execution thread, if any

property CurrentStep: TServiceMethodExecuteEventStep read fCurrentStep write fCurrentStep;
    The current state of the execution

property ExecutedInstancesFailed: TRawUTF8DynArray read fExecutedInstancesFailed;
    Contains exception serialization after ExecuteJson of multiple instances
    - follows the Instances[] order as supplied to RawExecute/ExecuteJson
    - if only a single Instances[] is supplied, the exception will be propagated to the caller, unless
      optIgnoreException option is defined
    - if more than one Instances[] is supplied, any raised Exception will be serialized using
      ObjectToJSONDebug(), or this property will be left to its default nil content if no exception
      occurred

property Input: TDocVariantData read fInput;
    Set if optInterceptInputOutput is defined in TServiceFactoryServer.Options
    - contains a dObject with input parameters as "argname":value pairs
    - this is a read-only property: you cannot change the input content

property LastException: Exception read fLastException;
    Only set during AddInterceptor() callback execution, if Step is smsError

property Method: PServiceMethod read fMethod;
    Low-level direct access to the associated method information

property OnCallback: TServiceMethodExecuteCallback read fOnCallback;
    Points e.g. to TSQLRestServerURIContext.ExecuteCallback

property Options: TServiceMethodOptions read fOptions write fOptions;
    Associated settings, as copied from TServiceFactoryServer.Options

property Output: TDocVariantData read fOutput;
    Set if optInterceptInputOutput is defined in TServiceFactoryServer.Options
    - contains a dObject with output parameters as "argname":value pairs
    - this is a read-only property: you cannot change the output content

property ServiceCustomAnswerHead: RawUTF8 read fServiceCustomAnswerHead write fServiceCustomAnswerHead;
    Set from output TServiceCustomAnswer.Header result parameter
property ServiceCustomAnswerStatus: cardinal read fServiceCustomAnswerStatus write fServiceCustomAnswerStatus;

Set from output TServiceCustomAnswer.Status result parameter

property Values: TPasserDynArray read fValues;

Low-level direct access to the current input/output parameter values
- you should not need to access this, but rather set optInterceptInputOutput in Options, and read Input/Output content

TServiceCustomAnswer = record
A record type to be used as result for a function method for custom content for interface-based services
- all answers are pure JSON object by default: using this kind of record as result will allow a response of any type (e.g. binary, HTML or text)
- this kind of answer will be understood by our TServiceContainerClient implementation, and it may be used with plain AJAX or HTML requests (via POST), to retrieve some custom content

Content: RawByteString;
The response body
- corresponding to the response type, as defined in Header

Header: RawUTF8;
Mandatory response type, as encoded in the HTTP header
- useful to set the response mime-type - see e.g. JSON_CONTENT_TYPE_HEADER_VAR TEXT_CONTENT_TYPE_HEADER or BINARY_CONTENT_TYPE_HEADER constants or GetMimeContentType() function
- in order to be handled as expected, this field SHALL be set to NOT " (otherwise TServiceCustomAnswer will be transmitted as raw JSON)

Status: cardinal;
The HTTP response code
- if not overriden, will default to HTTP_SUCCESS = 200 on server side
- on client side, will always contain HTTP_SUCCESS = 200 on success, or any error should be handled as expected by the caller (e.g. using TServiceFactoryClient.GetErrorMessage for decoding REST/HTTP errors)

TInterfaceResolver = class(TObject)
Abstract factory class allowing to call interface resolution in cascade
- you can inherit from this class to chain the TryResolve() calls so that several kind of implementations may be asked by a TInjectableObject, e.g. TInterfaceStub, TServiceObject or TDDDDRepositoryRestObjectMapping
- this will implement factory pattern, as a safe and thread-safe DI/IoC

TInterfaceResolverForSingleInterface = class(TInterfaceResolver)
Abstract factory class targetting a single kind of interface

constructor Create(const aInterface: TGUID; aImplementation: TInterfacedObjectClass); overload;

This overridden constructor will check and store the supplied class to implement an interface by TGUID
**constructor** Create(aInterface: PTypeInfo; aImplementation: TInterfacedObjectClass); overload;
   
   *This overridden constructor will check and store the supplied class to implement an interface*

**function** GetOneInstance(out Obj): boolean;
   
   *You can use this method to resolve the interface as a new instance*

**property** ImplementationClass: string read GetImplementationName;
   
   *The class name which will implement each repository instance*

**TInterfaceResolverInjected** = class(TInterfaceResolver)

   *Abstract factory class targeting any kind of interface*
   - you can inherit from this class to customize dependency injection (DI/IoC), defining the resolution via InjectStub/InjectResolver/InjectInstance methods, and doing the instance resolution using the overloaded Resolve*() methods
   - TServiceContainer will inherit from this class, as the main entry point for interface-based services of the framework (via TSQLRest.Services)
   - you can use RegisterGlobal() class method to define some process-wide DI

**destructor** Destroy; override;
   
   *Release all used instances*
   - including all TInterfacedObject instances as specified to Inject(aStubsByGUID)
   - will call _Release on all TInterfacedObject dependencies

**function** Resolve(aInterface: PTypeInfo; out Obj): boolean; overload;
   
   *Can be used to perform an DI/IoC for a given interface*
   - will search for the supplied interface to its internal list of resolvers
   - returns TRUE and set the Obj variable with a matching instance
   - can be used as such to resolve an ICalculator interface:
   
   ```pascal
   var calc: ICalculator;
   begin
   if Catalog.Resolve(TypeInfo(ICalculator),calc) then
   ... use calc methods
   ```

**function** Resolve(const aGUID: TGUID; out Obj): boolean; overload;

   *Can be used to perform an DI/IoC for a given interface*
   - you shall have registered the interface TGUID by a previous call to TInterfaceFactory.RegisterInterfaces([TypeInfo(ICalculator),...])
   - returns TRUE and set the Obj variable with a matching instance
   - can be used as such to resolve an ICalculator interface:
   
   ```pascal
   var calc: ICalculator;
   begin
   if ServiceContainer.Resolve(ICalculator,cal) then
   ... use calc methods
   ```

**procedure** InjectInstance(const aDependencies: array of TInterfacedObject); overload; virtual;

   *Prepare and setup interface DI/IoC resolution from a TInterfacedObject instance*
   - any TInterfacedObject declared as dependency will have its reference count increased, and decreased in Destroy
### InjectResolver

**Declaration:**
```delphi
procedure InjectResolver(const aOtherResolvers: array of TInterfaceResolver;
OwnOtherResolvers: boolean=false); overload; virtual;
```

**Purpose:**
Prepare and setup interface DI/IoC resolution with `TInterfaceResolver` kind of factory
- e.g. a customized `TInterfaceStub/TInterfaceMock`, a `TServiceContainer`, a `TDDDRepositoryRestObjectMapping` or any factory class
- by default, only `TInterfaceStub/TInterfaceMock` will be owned by this instance, and released by `Destroy` - unless you set `OwnOtherResolvers`

**Body:**
```delphi
procedure InjectStub(const aStubsByGUID: array of TGUID); overload; virtual;
```

**Purpose:**
Prepare and setup interface DI/IoC resolution with some blank `TInterfaceStub` specified by their `TGUID`

**RegisterGlobal**
```delphi
class procedure RegisterGlobal(aInterface: PTypeInfo; aImplementationClass: TInterfacedObjectClass); overload;
```

**Purpose:**
Define a global class type for interface resolution
- most of the time, you will need a local DI/IoC resolution list; but you may use this method to register a set of shared and global resolution patterns, common to the whole injection process
- by default, `TAutoLocker` and `TLockedDocVariant` will be registered by this unit to implement `IAutoLocker` and `ILockedDocVariant` interfaces

**RegisterGlobal**
```delphi
class procedure RegisterGlobal(aInterface: PTypeInfo; aImplementation: TInterfacedObject); overload;
```

**Purpose:**
Define a global instance for interface resolution
- most of the time, you will need a local DI/IoC resolution list; but you may use this method to register a set of shared and global resolution patterns, common to the whole injection process
- the supplied instance will be owned by the global list (incrementing its internal reference count), until it will be released via `RegisterGlobalDelete()`
- the supplied instance will be freed in the finalization of this unit, if not previously released via `RegisterGlobalDelete()`

**RegisterGlobalDelete**
```delphi
class procedure RegisterGlobalDelete(aInterface: PTypeInfo);
```

**Purpose:**
Undefine a global instance for interface resolution
- you can unregister a given instance previously defined via `RegisterGlobal(aInterface,aImplementation)`
- if you do not call `RegisterGlobalDelete()`, the remaining instances will be freed in the finalization of this unit

**Resolve**
```delphi
procedure Resolve(const aInterfaces: array of TGUID; const aObjs: array of pointer;
aRaiseExceptionIfNotFound: boolean=true); overload;
```

**Purpose:**
Can be used to perform several DI/IoC for a given set of interfaces
- here interfaces and instances are provided as `TGUID` and `@Instance`
- you shall have registered the interface `TGUID` by a previous call to `TInterfaceFactory.RegisterInterfaces([TypeInfo(ICalculator),...])`
- raise an `EServiceException` if any interface can't be resolved, unless `aRaiseExceptionIfNotFound` is set to FALSE
procedure ResolveByPair(const aInterfaceObjPairs: array of pointer; aRaiseExceptionIfNotFound: boolean=true);

Can be used to perform several DI/IoC for a given set of interfaces
- here interfaces and instances are provided as TypeInfo,@Instance pairs
- raise an EServiceException if any interface can't be resolved, unless aRaiseExceptionIfNotFound is set to FALSE

TInjectableObject = class(TInterfacedObjectWithCustomCreate)

Any service implementation class could inherit from this class to allow dependency injection aka SOLID DI/IoC by the framework
- once created, the framework will call AddResolver() member, so that its Resolve*() methods could be used to inject any needed dependency for lazy dependency resolution (e.g. within a public property getter)
- any interface published property will also be automatically injected
- if you implement a SOA service with this class, TSQLRestServer.Services will be auto-injected via TServiceFactoryServer.CreateInstance()

constructor CreateInjected(const aStubsByGUID: array of TGUID; const aOtherResolvers: array of TInterfaceResolver; const aDependencies: array of TInterfacedObject; aRaiseEServiceExceptionIfNotFound: boolean=true); virtual;

Initialize an instance, defining one or several mean of dependency resolution
- simple TInterfaceStub could be created directly from their TGUID, then any kind of DI/IoC resolver instances could be specified, i.e. either customized TInterfaceStub/TInterfaceMock, a TServiceContainer or a TDDDRepositoryRestObjectMapping, and then any TInterfacedObject instance will be used during dependency resolution:

procedure TMyTestCase.OneTestCaseMethod;
var
  Test: IServiceToBeTested;
begin
  Test := TServiceToBeTested.CreateInjected([ICalculator],
    [TInterfaceMock.Create(IPersistence,self).ExpectsCount('SaveItem',qoEqualTo,1),
      RestInstance.Services],
    [AnyInterfacedObject]);
...

- note that all the injected stubs/mocks instances will be owned by the TInjectableObject, and therefore released with it
- any TInterfacedObject declared as dependency will have its reference count increased, and decreased in Destroy
- once DI/IoC is defined, will call the AutoResolve() protected method

constructor CreateWithResolver(aResolver: TInterfaceResolver; aRaiseEServiceExceptionIfNotFound: boolean=true); virtual;

Initialize an instance, defining one dependency resolver
- the resolver may be e.g. a TServiceContainer
- once the DI/IoC is defined, will call the AutoResolve() protected method
- as called by TServiceFactoryServer.CreateInstance

destructor Destroy; override;

Release all used instances
- including all TInterfaceStub instances as specified to CreateInjected()
procedure Resolve(const aInterfaces: array of TGUID; const aObjs: array of pointer); overload;

  Can be used to perform several DI/IoC for a given set of interfaces
  - here interfaces and instances are provided as TGUID and pointers

procedure Resolve(aInterface: PTypeInfo; var Obj); overload;

  Can be used to perform a DI/IoC for a given interface type information

procedure Resolve(const aGUID: TGUID; var Obj); overload;

  Can be used to perform a DI/IoC for a given interface TGUID

procedure ResolveByPair(const aInterfaceObjPairs: array of pointer);

  Can be used to perform several DI/IoC for a given set of interfaces
  - here interfaces and instances are provided as TypeInfo,@Instance pairs

property Resolver: TInterfaceResolver read fResolver;

  Access to the associated dependency resolver, if any

TInjectableObjectRest = class(TInjectableObject)

  Service implementation class, with direct access on the associated
  TServiceFactoryServer/TSQLRestServer instances
  - allow dependency injection aka SOLID DI/IoC by the framework using inherited
    TInjectableObject.Resolve() methods
  - allows direct access to the underlying ORM using its Server method
  - this class will allow Server instance access outside the scope of remote SOA execution, e.g. when
    a DI is performed on server side: it is therefore a better alternative to ServiceContext.Factory,

constructor CreateWithResolverAndRest(aResolver: TInterfaceResolver; aFactory: TServiceFactoryServer; aServer: TSQLRestServer;
aRaiseEServiceExceptionIfNotFound: boolean=true); virtual;

  Initialize an instance, defining associated dependencies
  - the resolver may be e.g. a TServiceContainer
  - once the DI/IoC is defined, will call the AutoResolve() protected method
  - as called by   TServiceFactoryServer.CreateInstance

property Factory: TServiceFactoryServer read fFactory;

  Access to the associated interface factory
  - this property will be injected by TServiceFactoryServer.CreateInstance, so may be nil if the
    instance was created outside the SOA context

property Server: TSQLRestServer read fServer;

  Access to the associated REST Server, e.g. to its ORM methods
  - slightly faster than Factory.RestServer
  - this value will be injected by TServiceFactoryServer.CreateInstance, so may be nil if the
    instance was created outside the SOA context
IAutoCreateFieldsResolve = \texttt{interface}(IInterface)

*Used to set the published properties of a TInjectableAutoCreateFields*
- TInjectableAutoCreateFields.Create will check any resolver able to implement this interface, then run its SetProperties() method on it

\textbf{procedure} SetProperties(Instance: TObject);

*This method will be called once on any TInjectableAutoCreateFields just created instance*

TInjectableAutoCreateFields = \texttt{class}(TInjectableObject)

*Abstract class which will auto-inject its dependencies (DI/IoC), and also manage the instances of its TPersistent/TSynPersistent published properties*
- abstract class able with a virtual constructor, dependency injection (i.e. SOLID DI/IoC), and automatic memory management of all nested class published properties
- will also release any T*ObjArray dynamic array storage of persistents, previously registered via TJSONSerializer.RegisterObjArrayForJSON()
- this class is a perfect parent for any class storing data by value, and dependency injection, e.g. DDD services or daemons
- note that non published (e.g. public) properties won't be instantiated
- please take care that you will not create any endless recursion: you should ensure that at one level, nested published properties won't have any class instance matching its parent type
- since the destructor will release all nested properties, you should never store a reference of any of those nested instances outside
- if any associated resolver implements IAutoCreateFieldsResolve, its SetProperties() method will be called on all created TPersistent published properties, so that it may initialize its values

\textbf{constructor} Create; \texttt{override};

*This overridden constructor will instantiate all its nested TPersistent/TSynPersistent/TSynAutoCreateFields class published properties*
- then resolve and call IAutoCreateFieldsResolve.SetProperties(self)

\textbf{destructor} Destroy; \texttt{override};

*Finalize the instance, and release its published properties*

TInterfaceFactory = \texttt{class}(TObject)

*Class handling interface RTTI and fake implementation class*
- a thread-safe global list of such class instances is implemented to cache information for better speed: use class function TInterfaceFactory.Get() and not manual TInterfaceFactory.Create / Free
- if you want to search the interfaces by name or TGUID, call once Get(TypeInfo(IMyInterface)) or RegisterInterfaces() for proper registration
- will use TInterfaceFactoryRTTI classes generated from Delphi RTTI

\textbf{constructor} Create(aInterface: PTypeInfo);

*Initialize the internal properties from the supplied interface RTTI*
- it will check and retrieve all methods of the supplied interface, and prepare all internal structures for later use
- do not call this constructor directly, but TInterfaceFactory.Get()
function CheckMethodIndex(aMethodName: PUTF8Char): integer; overload;

*Find the index of a particular method in internal Methods[] list*
- won't find the default AddRef/Release/QueryInterface methods
- will raise an EInterfaceFactoryException if the method is not known

function CheckMethodIndex(const aMethodName: RawUTF8): integer; overload;

*Find the index of a particular method in internal Methods[] list*
- won't find the default AddRef/Release/QueryInterface methods
- will raise an EInterfaceFactoryException if the method is not known

function FindFullMethodIndex(const aFullMethodName: RawUTF8; alsoSearchExactMethodName: boolean=false): integer;

*Find the index of a particular interface.method in internal Methods[] list*
- will search for a match against Methods[].InterfaceDotMethodName property
- won't find the default AddRef/Release/QueryInterface methods
- will return -1 if the method is not known

function FindMethod(const aMethodName: RawUTF8): PServiceMethod;

*Find a particular method in internal Methods[] list*
- just a wrapper around FindMethodIndex() returning a PServiceMethod
- will return nil if the method is not known

function FindMethodIndex(const aMethodName: RawUTF8): integer;

*Find the index of a particular method in internal Methods[] list*
- will search for a match against Methods[].URI property
- won't find the default AddRef/Release/QueryInterface methods
- will return -1 if the method is not known
- if aMethodName does not have an exact method match, it will try with a trailing underscore,
  so that e.g. /service/start will match IService._Start()

class function Get(aInterface: PTypeInfo): TInterfaceFactory; overload;

*This is the main entry point to the global interface factory cache*
- access to this method is thread-safe
- this method will also register the class to further retrieval

class function Get(const aInterfaceName: RawUTF8): TInterfaceFactory; overload;

*Retrieve an interface factory from cache, from its name (e.g. 'IMyInterface')*
- access to this method is thread-safe
- you shall have registered the interface by a previous call to the overloaded Get(TypeInfo(IMyInterface)) method or RegisterInterfaces()
- if the supplied TGUID has not been previously registered, returns nil

class function Get(const aGUID: TGUID): TInterfaceFactory; overload;

*Retrieve an interface factory from cache, from its TGUID*
- access to this method is thread-safe
- you shall have registered the interface by a previous call to the overloaded Get(TypeInfo(IMyInterface)) method or RegisterInterfaces()
- if the supplied TGUID has not been previously registered, returns nil
function GetFullMethodName(aMethodIndex: integer): RawUTF8;

Returns the full 'Interface.MethodName' text, from a method index
- the method index should start at 0 for _free_/contract/_signature_ pseudo-methods, and
start at index 3 for real Methods[]
- will return plain 'Interface' text, if aMethodIndex is incorrect

function GetMethodName(MethodIndex: integer): RawUTF8;

Returns the method name from its method index
- the method index should start at 0 for _free_/contract/_signature_ pseudo-methods, and
start at index 3 for real Methods[]

class function GetUsedInterfaces: TSynObjectListLocked;

Returns the list of all declared TInterfaceFactory
- as used by SOA and mocking/stubbing features of this unit

class function GUID2TypeInfo(const aGUIDs: array of TGUID): PTypeInfoDynArray;

overload;

Could be used to retrieve an array of TypeInfo() from their GUID

class function GUID2TypeInfo(const aGUID: TGUID): PTypeInfo; overload;

Could be used to retrieve an array of TypeInfo() from their GUID

class procedure AddToObjArray(var Obj: TInterfaceFactoryObjArray; const aGUIDs: array of TGUID);

Add some TInterfaceFactory instances from their GUID

procedure CheckMethodIndexes(const aMethodName: array of RawUTF8; aSetAllIfNone: boolean; out aBits: TInterfaceFactoryMethodBits);

Set the Methods[] indexes bit from some methods names
- won't find the default AddRef/Release/QueryInterface methods
- will raise an EInterfaceFactoryException if the method is not known

class procedure RegisterInterfaces(const aInterfaces: array of PTypeInfo);

Register one or several interfaces to the global interface factory cache
- so that you can use TInterfaceFactory.Get(aGUID) or Get(aName)

class procedure RegisterUnsafeSPIType(const Types: array of pointer);

Register some TypeInfo() containing unsafe parameter values
- i.e. any RTTI type containing Sensitive Personal Information, e.g. a bank card number or a plain password
- such values will force associated values to be ignored during logging, as a more tuned alternative to optNoLogInput or optNoLogOutput

property DocVariantOptions: TDocVariantOptions read fDocVariantOptions write fDocVariantOptions;

How this interface will work with variants (including TDocVariant)
- by default, contains JSON_OPTIONS_FAST for best performance - i.e.
  [dvoReturnNullForUnknownProperty,dvoValueCopiedByReference]

property InterfaceIID: TGUID read fInterfaceIID;

The registered Interface GUID
property InterfaceName: RawUTF8 read fInterfaceName;
  Will return the interface name, e.g. 'ICalculator'
  - published property to be serializable as JSON e.g. for debugging info

property InterfaceTypeInfo: PTypeInfo read fInterfaceTypeInfo;
  The registered Interface low-level Delphi RTTI type

property InterfaceURI: RawUTF8 read fInterfaceURI write fInterfaceURI;
  The interface name, without its initial 'I'
  - e.g. ICalculator -> 'Calculator'

property MethodIndexCallbackReleased: Integer read fMethodIndexCallbackReleased;
  Identifies a CallbackReleased() method in this interface
  - i.e. the index in Methods[] of the following signature:
    procedure CallbackReleased(const callback: IInvokable; const interfaceName: RawUTF8);
  - this method will be called e.g. by TInterfacedCallback.Destroy, when a callback is released on
    the client side so that you may be able e.g. to unsubscribe the callback from an interface list (via
    InterfaceArrayDelete)
  - contains -1 if no such method do exist in the interface definition

property MethodIndexCurrentFrameCallback: Integer read fMethodIndexCurrentFrameCallback;
  Identifies a CurrentFrame() method in this interface
  - i.e. the index in Methods[] of the following signature:
    procedure CurrentFrame(isLast: boolean);
  - this method will be called e.g. by TSQLHttpClientWebsockets.CallbackRequest for interface
    callbacks in case of WebSockets jumbo frames, to allow e.g. faster database access via a batch
  - contains -1 if no such method do exist in the interface definition

property Methods: TServiceMethodDynArray read fMethods;
  The declared internal methods
  - list does not contain default AddRef/Release/QueryInterface methods
  - nor the _free_/_contract_/ _signature_ pseudo-methods

property MethodsCount: cardinal read fMethodsCount;
  The number of internal methods
  - does not include the default AddRef/Release/QueryInterface methods
  - nor the _free_/_contract_/ _signature_ pseudo-methods

TInterfaceFactoryRTTI = class(TInterfaceFactory)
  Class handling interface RTTI and fake implementation class
  - this class only exists for Delphi 6 and up, since FPC does not generate the expected RTTI - see
    http://bugs.freepascal.org/view.php?id=26774

TInterfacedObjectFromFactory = class(TInterfacedObject)
  Abstract class handling a generic interface implementation class
  constructor Create(aFactory: TInterfaceFactory; aOptions:
    TInterfacedObjectFromFactoryOptions; aInvoke: TOnFakeInstanceInvoke;
    aNotifyDestroy: TOnFakeInstanceDestroy);
  Create an instance, using the specified interface
destructor Destroy; override;
  Release the remote server instance (in sicClientDriven mode);

property ClientDrivenID: Cardinal read fClientDrivenID;
  The ID used in sicClientDriven mode

property Factory: TInterfaceFactory read fFactory;
  The associated interface factory class

TInterfaceFactoryGenerated = class(TInterfaceFactory)
Class handling interface implementation generated from source
- this class targets oldest FPC, which did not generate the expected RTTI - see
  http://bugs.freepascal.org/view.php?id=26774
- mORMotWrapper.pas will generate a new inherited class, overriding abstract
  AddMethodsFromTypeInfo() to define the interface methods

class procedure RegisterInterface(aInterface: PTypeInfo); virtual;
  Register one interface type definition from the current class
  - will be called by mORMotWrapper.pas generated code, in initialization section, so that the
    needed type information will be available

TOnInterfaceStubExecuteParamsAbstract = class(TObject)
Abstract parameters used by TInterfaceStub.Executes() events callbacks

constructor Create(aSender: TInterfaceStub; aMethod: PServiceMethod; const
  aParams,aEventParams: RawUTF8); virtual;
  Constructor of one parameters marshalling instance

procedure Error(const Format: RawUTF8; const Args: array of const); overload;
  Call this method if the callback implementation failed

procedure Error(const aErrorMessage: RawUTF8); overload;
  Call this method if the callback implementation failed

property EventParams: RawUTF8 read fEventParams;
  A custom message, defined at TInterfaceStub.Executes() definition

property Failed: boolean read fFailed;
  Low-level flag, set to TRUE if one of the Error() method was called

property Method: PServiceMethod read fMethod;
  Pointer to the method which is to be executed

property Result: RawUTF8 read fResult;
  Outgoing values array encoded as JSON
  - every var, out parameter or the function result shall be encoded as a JSON array into this
    variable, in the same order than the stubbed method declaration
  - use Returns() method to create the JSON array directly, from an array of values

property Sender: TInterfaceStub read fSender;
  The stubbing / mocking generator
property TestCase: TSynTestCase read GetSenderAsMockTestCase;

The mocking generator associated test case
- will raise an exception if the associated Sender generator is not a TInterfaceMock

TOOnInterfaceStubExecuteParamsVariant =
class(TOnInterfaceStubExecuteParamsAbstract)

Parameters used by TInterfaceStub.Executes() events callbacks as Variant
- this class will expect input and output parameters to specified as variant arrays properties, so is
easier (and a bit slower) than the TOnInterfaceStubExecuteParamsJSON class

constructor Create(aSender: TInterfaceStub; aMethod: PServiceMethod; const
aParams,aEventParams: RawUTF8); override;

Constructor of one parameters marshalling instance

function InputAsDocVariant(Kind: TServiceMethodParamsDocVariantKind; Options:
TDocVariantOptions=[dvoReturnNullForUnknownProperty,dvoValueCopiedByReference]):
variant;

Returns the input parameters as a TDocVariant object or array

function OutputAsDocVariant(Kind:
TServiceMethodParamsDocVariantKind; Options:
TDocVariantOptions=[dvoReturnNullForUnknownProperty,dvoValueCopiedByReference]):
variant;

Returns the output parameters as a TDocVariant object or array

procedure AddLog(aLog: TSynLogClass; aOutput: boolean; aLevel:
TSynLogInfo=sllTrace);

Log the input or output parameters to a log instance

property Input[Index: Integer]: variant read GetInput;

Input parameters when calling the method
- order shall follow the method const and var parameters
- if the supplied Index is out of range, an EInterfaceStub will be raised

property Named[const ParamName: RawUTF8]: variant read GetInNamed write SetOutNamed;

Access to input/output parameters when calling the method
- if the supplied name is incorrect, an EInterfaceStub will be raised
- is a bit slower than Input[]/Output[] indexed properties, but easier to work with, and safer in
case of method signature change (like parameter add or rename)
- marked as default property, so you can use it e.g. as such:
  procedure TFooTestCase.ExecuteBar(Ctxt: TOnInterfaceStubExecuteParamsVariant);
  begin
  Ctxt['i'] := Ctxt['i']+1;  // i := i+1;
  Ctxt['result'] := 42;     // result := 42;
  end;

to emulate this native implementation:
  function Bar(var i: Integer): Integer;
  begin
    inc(i);
    result := 42;
  end;

- using this default Named[] property is recommended over the index-based Output[] property
- if an Output[]/Named[] item is not set, a default value will be used
**property** Output[Index: Integer]: variant write SetOutput;

*Output parameters returned after method process*
- order shall follow the method var, out parameters and the function result (if method is not a procedure)
- if the supplied Index is out of range, an EInterfaceStub will be raised
- can be used as such:

```pascal
procedure TFooTestCase.ExecuteBar(Ctxt: TOnInterfaceStubExecuteParamsVariant);
begin
  // Input[0]=i
  Ctxt.Output[0] := Ctxt.Input[0]+1; // i := i+1;
  Ctxt.Output[1] := 42; // result := 42;
end; // Output[0]=i, Output[1]=result
```

to emulate this native implementation:
```pascal
function Bar(var i: Integer): Integer;
begin
  inc(i);
  result := 42;
end;
```

- consider using the safest Named[] property, to avoid parameters index matching issue
- if an Output[]/Named[] item is not set, a default value will be used

**property** UTF8[const ParamName: RawUTF8]: RawUTF8 read GetInUTF8;

*Access to UTF-8 input parameters when calling the method*
- if the supplied name is incorrect, an EInterfaceStub will be raised
- is a bit slower than Input[]/Output[] indexed properties, but easier to work with, and safer in case of method signature change (like parameter add or rename)
- slightly easier to use Ctxt.UTF8['str'] than ToUTF8(Ctxt.Named['str'])

```pascal
TOnInterfaceStubExecuteParamsJSON = class(TOnInterfaceStubExecuteParamsAbstract)

  *Parameters used by TInterfaceStub.Executes() events callbacks as JSON*
  - this class will expect input and output parameters to be encoded as JSON arrays, so is faster than TOnInterfaceStubExecuteParamsVariant

  **procedure** Returns(const ValuesJsonArray: RawUTF8); overload;
  
  *A method to return a JSON array of values into Result*
  - expected format is e.g. '[43,42]'

  **procedure** Returns(const Values: array of const); overload;
  
  *A method to return an array of values into Result*
  - just a wrapper around JSONEncodeArrayOfConst([...])

  can be used as such:

  ```pascal
  procedure TFooTestCase.ExecuteBar(var Ctxt: TOnInterfaceStubExecuteParamsJSON);
  begin
    // Ctxt.Params := 'i' -> Ctxt.Result := '[i+1,42]' 
    Ctxt.Returns([GetInteger(pointer(Ctxt.Params))+1,42]);
  end;
  ```
```

to emulate this native implementation:
```pascal
function Bar(var i: Integer): Integer;
begin
  inc(i);
  result := 42;
end;
```
property Params: RawUTF8 read fParams;

  Incoming parameters array encoded as JSON array without braces
  - order follows the method const and var parameters
  
  Stub.Add(10, 20) -> Params = '10,20';

TInterfaceStubRule = record

  Define a mocking / stubing rule used internally by TInterfaceStub

  ExceptionClass: ExceptClass;
  
  The exception class to be raised
  - for TInterfaceStub.Raises(), Values contains Exception.Message

  Execute: TMethod;
  
  The event handler to be executed
  - for TInterfaceStub.Executes(), Values is transmitted as aResult parameter
  - either a TOnInterfaceStubExecuteJSON, or a TOnInterfaceStubExecuteVariant

  ExpectedPassCount: cardinal;
  
  Expected pass count value set by TInterfaceStub.ExpectsCount()
  - value to be compared to the number of times this rule has been executed
  - TInterfaceStub/TInterfaceMock will check it in their Destroy destructor, using the comparison
    stated by ExpectedPassCountOperator

  ExpectedPassCountOperator: TSQLQueryOperator;
  
  Comparison operator set by TInterfaceStub.ExpectsCount()
  - only qoEqualTo..qoGreaterThanOrEqualTo are relevant here

  ExpectedTraceHash: cardinal;
  
  Log trace value set by TInterfaceStub.ExpectsTrace()
  - value to be compared to the Hash32() value of the execution log trace
  - TInterfaceStub/TInterfaceMock will check it in their Destroy destructor, using the fLogs[]
    content

  Kind: TInterfaceStubRuleKind;
  
  The type of this rule
  - isUndefined is used for a TInterfaceStub.ExpectsCount() weak rule

  Params: RawUTF8;
  
  Optional expected parameters, serialized as a JSON array
  - if equals '', the rule is not parametrized - i.e. it will be the default for this method

  RulePassCount: cardinal;
  
  The number of times this rule has been executed
Values: RawUTF8;

Values associated to the rule
- for TInterfaceStub.Executes(), is the aEventParams parameter transmitted to Execute event handler (could be used to e.g. customize the handler)
- for TInterfaceStub.Raises(), is the Exception.Message associated to one ExceptionClass
- for TInterfaceStub.Returns(), is the returned result, serialized as a JSON array (including var / out parameters then any function result)
- for TInterfaceStub.Fails() is the returned error message for TInterfaceStub exception or TInterfaceMock associated test case

TInterfaceStubRules = object(TObject)
Define the rules for a given method as used internaly by TInterfaceStub

DefaultRule: integer;
Index in Rules[] of the default rule, i.e. the one with Params=""

MethodPassCount: cardinal;
The number of times this method has been executed

Rules: array of TInterfaceStubRule;
The mocking / stubing rules associated to this method

function FindRuleIndex(const aParams: RawUTF8): integer;
Find a rule index from its Params content

function FindStrongRuleIndex(const aParams: RawUTF8): integer;
Find a strong rule index from its Params content

procedure AddRule(Sender: TInterfaceStub; aKind: TInterfaceStubRuleKind; const aParams, aValues: RawUTF8; const aEvent: TNotifyEvent=nil; aExceptionClass: ExceptClass=nil; aExpectedPassCountOperator: TSQLQueryOperator=qoNone; aValue: cardinal=0);
Register a rule

TInterfaceStubLog = object(TObject)
Used to keep track of one stubbed method call

CustomResults: RawUTF8;
Any non default result returned after execution
- if not set (i.e. if equals ""), Method^.DefaultResult has been returned
- if WasError is TRUE, always contain the error message

Method: PServiceMethod;
The method called
- a pointer to the existing information in shared TInterfaceFactory

Params: RawUTF8;
The parameters at execution call, as JSON CSV (i.e. array without [ ])

mORMot.pas unit - Rev. 1.18
**Timestamp64**: Int64;

*Call timestamp, in milliseconds*
- is filled with GetTickCount64() API returned value

**WasError**: boolean;

*Set to TRUE if this calls failed*
- i.e. if EInterfaceFactoryException was raised for TInterfaceStub, or if TInterfaceMock did notify its associated TSynTestCase via a Check()
- CustomResults/Results will contain the error message

**function** Results: RawUTF8;

*The result returned after execution*
- this method will return Method^.DefaultResult if CustomResults="

**procedure** AddAsText(WR: TTextWriter; aScope: TInterfaceStubLogLayouts; SepChar: AnsiChar=',');

*Append the log in textual format*
- typical output is as such:
  Add(10,20)=[30],
  or, if WasError is TRUE:
  Divide(20,0) error "divide by zero",

**TInterfaceStub** = **class**(TInterfaceResolver)

*Used to stub an interface implementation*
- define the expected workflow in a fluent interface using Executes / Fails / Returns / Raises
- this class will be inherited by TInterfaceMock which will contain some additional methods dedicated to mocking behavior (e.g. including in tests)
- each instance of this class will be owned by its generated fake implementation class (retrieved at constructor out parameter): when the stubed/mocked interface is freed, its associated TInterfaceStub will be freed - so you do not need to protect TInterfaceStub.Create with a try..finally clause, since it will be released when no more needed
- inherits from TInterfaceResolver so match TInjectableObject expectations

**constructor** Create(const aInterfaceName: RawUTF8; out aStubbedInterface);

*Initialize an interface stub from an interface name (e.g. 'IMyInterface'*
- you shall have registered the interface by a previous call to TInterfaceFactory.Get(TTypeInfo(IMyInterface)) or RegisterInterfaces([])
- if the supplied name has not been previously registered, raise an Exception

**constructor** Create(aInterface: PTypeInfo); **reintroduce**; overload;

*Prepare an interface stub from TypeInfo(IMyInterface) for later injection*
- create several TInterfaceStub instances for a given TInjectableObject

**procedure** TMyTestCase.OneTestCaseMethod;

*var* Test: IServiceToBeTested;

*begin*
  Test := TServiceToBeTested.CreateInjected([],
  TInterfaceStub.Create(TTypeInfo(ICalculator)),
  TInterfaceMock.Create(TTypeInfo(IPersistence), self).
  ExpectsCount('SaveItem', qoEqualTo, 1));
**constructor** Create(const aGUID: TGUID); reintroduce; overload;

Prepare an interface stub from a given TGUID for later injection
- you shall have registered the interface by a previous call to
  TInterfaceFactory.RegisterInterfaces([TypeInfo(IMyInterface),...])
- then create TInterfaceStub instances for a given TInjectableObject:

```pascal
procedure TMyTestCase.OneTestCaseMethod;
var Test: IServiceToBeTested;
beg
  Test := TServiceToBeTested.CreateInjected([IMyInterface],
    TInterfaceMock.Create(IPersistence,self).
  ExpectsCount('SaveItem',qoEqualTo,1));
end;
```

**constructor** Create(aFactory: TInterfaceFactory; const aInterfaceName: RawUTF8); reintroduce; overload; virtual;

Low-level internal constructor
- you should not call this method, but the overloaded alternatives

**constructor** Create(aInterface: PTypeInfo; out aStubbedInterface); reintroduce; overload;

Initialize an interface stub from TypeInfo(IMyInterface)
- assign the fake class instance to a stubbed interface variable:

```pascal
var I: ICalculator;
TInterfaceStub.Create(TypeInfo(ICalculator),I);
Check(I.Add(10,20)=0,'Default result');
```

**constructor** Create(const aGUID: TGUID; out aStubbedInterface); reintroduce; overload;

Initialize an interface stub from an interface GUID
- you shall have registered the interface by a previous call to
  TInterfaceFactory.RegisterInterfaces([TypeInfo(IMyInterface),...])
- once registered, create and use the fake class instance as such:

```pascal
var I: ICalculator;
TInterfaceStub.Create(ICalculator,I);
Check(I.Add(10,20)=0,'Default result');
```

- if the supplied TGUID has not been previously registered, raise an Exception

**function** Executes(const aMethodName: RawUTF8; const aParams: array of const; aEvent: TOnInterfaceStubExecuteVariant; const aEventParams: RawUTF8=''): TInterfaceStub; overload;

Add an execution rule for a given method and a set of parameters, with Variant marshalling
- if execution context matches the supplied aParams value, aEvent is triggered
  - optional aEventParams parameter will be transmitted to aEvent handler
  - raise an Exception if the method name does not exist for this interface

**function** Executes(const aMethodName, aParams: RawUTF8; aEvent: TOnInterfaceStubExecuteVariant; const aEventParams: RawUTF8=''): TInterfaceStub; overload;

Add an execution rule for a given method and a set of parameters, with Variant marshalling
- if execution context matches the supplied aParams value, aEvent is triggered
  - optional aEventParams parameter will be transmitted to aEvent handler
  - raise an Exception if the method name does not exist for this interface

Will add execution rules for all methods to log the input parameters
- aKind will define how the input parameters are serialized in JSON

function Executes(aEvent: TOnInterfaceStubExecuteVariant; const aEventParams: RawUTF8=''): TInterfaceStub; overload;

Add an execution rule for all methods, with Variant marshalling
- optional aEventParams parameter will be transmitted to aEvent handler
- callback's Cxt: TOnInterfaceStubExecuteParamsVariant's Method field will identify the executed method

function Executes(const aMethodName, aParams: RawUTF8; aEvent: TOnInterfaceStubExecuteJSON; const aEventParams: RawUTF8=''): TInterfaceStub; overload;

Add an execution rule for a given method and a set of parameters, with JSON marshalling
- if execution context matches the supplied aParams value, aEvent is triggered
- optional aEventParams parameter will be transmitted to aEvent handler
- raise an Exception if the method name does not exist for this interface

function Executes(const aMethodName: RawUTF8; aEvent: TOnInterfaceStubExecuteVariant; const aEventParams: RawUTF8=''): TInterfaceStub; overload;

Add an execution rule for a given method, with Variant marshalling
- optional aEventParams parameter will be transmitted to aEvent handler
- raise an Exception if the method name does not exist for this interface

function Executes(const aMethodName: RawUTF8; const aParams: array of const; aEvent: TOnInterfaceStubExecuteJSON; const aEventParams: RawUTF8=''): TInterfaceStub; overload;

Add an execution rule for a given method and a set of parameters, with JSON marshalling
- if execution context matches the supplied aParams value, aEvent is triggered
- optional aEventParams parameter will be transmitted to aEvent handler
- raise an Exception if the method name does not exist for this interface

function ExpectsCount(const aMethodName: RawUTF8; const aParams: array of const; aOperator: TSQLQueryOperator; aValue: cardinal): TInterfaceStub; overload;

Add a pass count expectation rule for a given method and a set of parameters
- those rules will be evaluated at Destroy execution
- only qoEqualTo..qoGreaterThanOrEqualTo are relevant here
- it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case
- raise an Exception if the method name does not exist for this interface
function ExpectsCount(const aMethodName: RawUTF8; aOperator: TSQLQueryOperator; aValue: cardinal): TInterfaceStub; overload;
    Add a pass count expectation rule for a given method
    - those rules will be evaluated at Destroy execution
    - only qoEqualTo..qoGreaterThanOrEqualTo are relevant here
    - it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case
    - raise an Exception if the method name does not exist for this interface

function ExpectsCount(const aMethodName, aOperator: TSQLQueryOperator; aValue: cardinal): TInterfaceStub; overload;
    Add a pass count expectation rule for a given method and a set of parameters
    - those rules will be evaluated at Destroy execution
    - only qoEqualTo..qoGreaterThanOrEqualTo are relevant here
    - it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case
    - raise an Exception if the method name does not exist for this interface

function ExpectsTrace(const aMethodName: RawUTF8; aValue: RawUTF8): TInterfaceStub; overload;
    Add a JSON-based execution expectation rule for a given method
    - those rules will be evaluated at Destroy execution
    - supplied aValue is the trace in LogAsText format
    - it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case
    - raise an Exception if the method name does not exist for this interface

function ExpectsTrace(const aMethodName, aValue: RawUTF8): TInterfaceStub; overload;
    Add a hash-based execution expectation rule for a given method
    - those rules will be evaluated at Destroy execution
    - supplied aValue is a Hash32() of the trace in LogAsText format
    - it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case
    - raise an Exception if the method name does not exist for this interface

function ExpectsTrace(aValue: cardinal): TInterfaceStub; overload;
    Add a hash-based execution expectation rule for the whole interface
    - those rules will be evaluated at Destroy execution
    - supplied aValue is a Hash32() of the trace in LogAsText format
    - it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case

function ExpectsTrace(const aMethodName, aParams: RawUTF8; aValue: cardinal): TInterfaceStub; overload;
    Add a hash-based execution expectation rule for a given method and a set of parameters
    - those rules will be evaluated at Destroy execution
    - supplied aValue is a Hash32() of the trace in LogAsText format
    - it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case
    - raise an Exception if the method name does not exist for this interface
function ExpectsTrace(const aMethodName: RawUTF8; const aParams: array of const; const aValue: RawUTF8): TInterfaceStub; overload;

*Add a JSON-based execution expectation rule for a given method and a set of parameters*
- those rules will be evaluated at Destroy execution
- supplied aValue is the trace in LogAsText format
- it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case
- raise an Exception if the method name does not exist for this interface

function ExpectsTrace(const aValue: RawUTF8): TInterfaceStub; overload;

*Add a JSON-based execution expectation rule for the whole interface*
- those rules will be evaluated at Destroy execution
- supplied aValue is the trace in LogAsText format
- it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case

function ExpectsTrace(const aMethodName: RawUTF8; const aParams: array of const; aValue: cardinal): TInterfaceStub; overload;

*Add a hash-based execution expectation rule for a given method and a set of parameters*
- those rules will be evaluated at Destroy execution
- supplied aValue is a Hash32() of the trace in LogAsText format
- it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case
- raise an Exception if the method name does not exist for this interface

function ExpectsTrace(const aMethodName, aParams, aValue: RawUTF8): TInterfaceStub; overload;

*Add a JSON-based execution expectation rule for a given method and a set of parameters*
- those rules will be evaluated at Destroy execution
- supplied aValue is the trace in LogAsText format
- it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case
- raise an Exception if the method name does not exist for this interface

function Fails(const aMethodName, aErrorMsg: RawUTF8): TInterfaceStub; overload;

*Add an error rule for a given method*
- an error will be returned to the caller, with aErrorMsg as message
- it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case
- raise an Exception if the method name does not exist for this interface

function Fails(const aMethodName, aParams, aErrorMsg: RawUTF8): TInterfaceStub; overload;

*Add an error rule for a given method and a set of parameters*
- an error will be returned to the caller, with aErrorMsg as message
- it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case
- raise an Exception if the method name does not exist for this interface
function Fails(const aMethodName: RawUTF8; const aParams: array of const; const aErrorMsg: RawUTF8): TInterfaceStub; overload;
    Add an error rule for a given method and a set of parameters
    - an error will be returned to the caller, with aErrorMsg as message
    - it will raise EInterfaceFactoryException for TInterfaceStub, but TInterfaceMock will push the failure to the associated test case
    - raise an Exception if the method name does not exist for this interface

function LogAsText(SepChar: AnsiChar='\',\'): RawUTF8;
    The stubbed method execution trace converted as text
    - typical output is a list of calls separated by commas:
      Add(10,20)=[30],Divide(20,0) error "divide by zero"

function Raises(const aMethodName: RawUTF8; aException: ExceptClass; const aMessage: string): TInterfaceStub; overload;
    Add an exception rule for a given method
    - will create and raise the specified exception for this method
    - raise an Exception if the method name does not exist for this interface

function Raises(const aMethodName: RawUTF8; const aParams: RawUTF8; aException: ExceptClass; const aMessage: string): TInterfaceStub; overload;
    Add an exception rule for a given method and a set of parameters
    - will create and raise the specified exception for this method, if the execution context matches the supplied aParams value
    - raise an Exception if the method name does not exist for this interface

function Raises(const aMethodName: RawUTF8; const aParams: array of const; aException: ExceptClass; const aMessage: string): TInterfaceStub; overload;
    Add an exception rule for a given method and a set of parameters
    - will create and raise the specified exception for this method, if the execution context matches the supplied aParams value
    - raise an Exception if the method name does not exist for this interface

function Returns(const aMethodName: RawUTF8; const aExpectedResults: array of const): TInterfaceStub; overload;
    Add an evaluation rule for a given method
    - aExpectedResults will be returned to the caller after conversion to a JSON array
    - raise an Exception if the method name does not exist for this interface

function Returns(const aMethodName, aExpectedResults: RawUTF8): TInterfaceStub;
    overload;
    Add an evaluation rule for a given method
    - aExpectedResults JSON array will be returned to the caller
    - raise an Exception if the method name does not exist for this interface

function Returns(const aMethodName: RawUTF8; const aParams, aExpectedResults: array of const): TInterfaceStub; overload;
    Add an evaluation rule for a given method and a set of parameters
    - aExpectedResults JSON array will be returned to the caller
    - raise an Exception if the method name does not exist for this interface
function Returns(const aMethodName, aParams, aExpectedResults: RawUTF8): TInterfaceStub; overload;

   Add an evaluation rule for a given method and a set of parameters
   - aExpectedResults JSON array will be returned to the caller
   - raise an Exception if the method name does not exist for this interface

function SetOptions(Options: TInterfaceStubOptions): TInterfaceStub;

   Set the optional stubing/mocking options
   - same as the Options property, but in a fluent-style interface

procedure ClearLog;

   Reset the internal trace
   - Log, LogAsText, LogHash and LogCount will be initialized

property InterfaceFactory: TInterfaceFactory read fInterface;

   Access to the registered Interface RTTI information

property LastInterfacedObjectFake: TInterfacedObject read fLastInterfacedObjectFake;

   Returns the last created TInterfacedObject instance
   - e.g. corresponding to the out aStubbedInterface parameter of Create()

property Log: TInterfaceStubLogDynArray read fLogs;

   The stubbed method execution trace items

property LogCount: Integer read fLogCount;

   The stubbed method execution trace number of items

property LogHash: cardinal read GetLogHash;

   The stubbed method execution trace converted as one numerical hash
   - returns Hash32(LogAsText)

property Options: TInterfaceStubOptions read fOptions write IntSetOptions;

   Optional stubing/mocking options
   - you can use the SetOptions() method in a fluent-style interface

TInterfaceMock = class(TInterfaceStub)

   Used to mock an interface implementation via expect-run-verify pattern
   - TInterfaceStub will raise an exception on Fails(), ExpectsCount() or ExpectsTrace() rule activation,
     but TInterfaceMock will call TSynTestCase.Check() with no exception with such rules, as expected
     by a mocked interface
   - this class will follow the expect-run-verify pattern, i.e. expectations are defined before running
     the test, and verification is performed when the instance is released - use TInterfaceMockSpy if
     you prefer the more explicit run-verify pattern

constructor Create(aInterface: PTypeInfo; aTestCase: TSynTestCase); reintroduce;

   Initialize an interface mock from TypeInfo(IMyInterface) for later injection
   - aTestcase.Check() will be called in case of mocking failure
**constructor** Create(const aGUID: TGUID; aTestCase: TSynTestCase); reintroduce; overload;

*Initialize an interface mock from TypeInfo(IMyInterface) for later injection*
- aTestCase.Check() will be called in case of mocking failure

**constructor** Create(const aInterfaceName: RawUTF8; out aMockedInterface; aTestCase: TSynTestCase); reintroduce; overload;

*Initialize an interface mock from an interface name (e.g. 'IMyInterface')*
- aTestCase.Check() will be called in case of mocking failure
- you shall have registered the interface by a previous call to TInterfaceFactory.GetTypeInfo(IMyInterface) or RegisterInterfaces()
- if the supplied name has not been previously registered, raise an Exception

**constructor** Create(aInterface: PTypeInfo; out aMockedInterface; aTestCase: TSynTestCase); reintroduce; overload;

*Initialize an interface mock from TypeInfo(IMyInterface)*
- aTestCase.Check() will be called in case of mocking failure
- you shall have registered the interface by a previous call to TInterfaceFactory.RegisterInterfaces([TypeInfo(IPersistence),...])
- once registered, create and use the fake class instance as such:
  ```pascal
  TMyTestCase.OneTestCaseMethod;

  var Persist: IPersistence;
  ...
  TInterfaceMock.CreateTypeInfo(IPersistence),Persist,self).
  ExpectsCount('SaveItem',qoEqualTo,1));
  ```

**constructor** Create(const aGUID: TGUID; out aMockedInterface; aTestCase: TSynTestCase); reintroduce; overload;

*Initialize an interface mock from an interface TGUID*
- aTestCase.Check() will be called during validation of all Expects*()
- you shall have registered the interface by a previous call to TInterfaceFactory.RegisterInterfaces([TypeInfo(IPersistence),...])
- if the supplied TGUID has not been previously registered, raise an Exception

**property** TestCase: TSynTestCase read fTestCase;

*The associated test case*

TInterfaceMockSpy = class(TInterfaceMock)

*Used to mock an interface implementation via run-verify pattern*
- this class will implement a so called "test-spy" mocking pattern, i.e. no expectation is to be declared at first, but all calls are internally logged (i.e. it force imoLogMethodCallsAndResults option to be defined), and can afterwards been check via Verify() calls

**constructor** Create(aFactory: TInterfaceFactory; const aInterfaceName: RawUTF8); override;

*This will set and force imoLogMethodCallsAndResults option as needed*
- you should not call this method, but the overloaded alternatives
procedure Verify(const aMethodName, aTrace: RawUTF8; aScope: TInterfaceMockSpyCheck); overload;

Check an execution trace for a specified method
- text trace format will follow specified scope, e.g.
  Verify('Add','(10,30),(2,35)',chkNameParams);

or include parameters and function results:
  Verify('Add','(10,30)=[300],(2,35)=[37]',chkNameParamsResults);
- if aMethodName does not exists or aScope=chkName, will raise an exception

procedure Verify(const aMethodName, aParams, aTrace: RawUTF8); overload;

Check an execution trace for a specified method and parameters
- text trace format shall contain only results, e.g.
  Verify('Add','2,35','[37]');

procedure Verify(const aMethodName: RawUTF8; const aParams: array of const; const aTrace: RawUTF8); overload;

Check an execution trace for a specified method and parameters
- text trace format shall contain only results, e.g.
  Verify('Add',[2,35],[37]');

procedure Verify(const aTrace: RawUTF8; aScope: TInterfaceMockSpyCheck); overload;

Check an execution trace for the global interface
- text trace format shall follow method calls, e.g.
  Verify('Multiply,Add',chkName);

or may include parameters:
  Verify('Multiply(10,30),Add(2,35)',chkNameParams);

or include parameters and function results:
  Verify('Multiply(10,30)=[300],Add(2,35)=[37]',chkNameParamsResults);

procedure Verify(const aMethodName: RawUTF8; aOperator: TSQLQueryOperator=qoGreaterThan; aCount: cardinal=0); overload;

Check that a method has been called a specify number of times

procedure Verify(const aMethodName, aParams: RawUTF8; aOperator: TSQLQueryOperator=qoGreaterThan; aCount: cardinal=0); overload;

Check a method calls count with a set of parameters
- parameters shall be defined as a JSON array of values

procedure Verify(const aMethodName: RawUTF8; const aParams: array of const; aOperator: TSQLQueryOperator=qoGreaterThan; aCount: cardinal=0); overload;

Check a method calls count with a set of parameters
- parameters shall be defined as a JSON array of values
The purpose of TServiceFactory is to provide an abstract service provider, as registered in TServiceContainer:

- each registered interface has its own TServiceFactory instance, available as one TSQLServiceContainer item from TSQLRest.Services property
- this will be either implemented by a registered TInterfacedObject on the server, or by a on-the-fly generated fake TInterfacedObject class communicating via JSON on a client
- TSQLRestServer will have to register an interface implementation as:
  ```
  Server.ServiceRegister(TServiceCalculator,[TypeInfo(ICalculator)],sicShared);
  ```
- TSQLRestClientURI will have to register an interface remote access as:
  ```
  Client.ServiceRegister([TypeInfo(ICalculator)],sicShared));
  ```

Note that the implementation (TServiceCalculator) remain on the server side only: the client only needs the ICalculator interface.

- then TSQLRestServer and TSQLRestClientURI will both have access to the service, via their Services property, e.g. as:

  ```
  var I: ICalculator;
  ...
  if Services.Info(ICalculator).Get(I) then
    result := I.Add(10,20);
  ```

Which is in practice to be used with the faster wrapper method:

  ```
  if Services.Resolve(ICalculator,I) then
    result := I.Add(10,20);
  ```

Used for DI-2.1.5 (page 2547).

**Constructor** Create(aRest: TSQLRest; aInterface: PTypeInfo; aInstanceCreation: TServiceInstanceImplementation; const aContractExpected: RawUTF8);

- Initialize the service provider parameters
  - it will check and retrieve all methods of the supplied interface, and prepare all internal structures for its serialized execution

**Function** Get(out Obj): Boolean; virtual; abstract;

- Retrieve an instance of this interface
  - this virtual method will be overridden to reflect the expected behavior of client or server side
  - can be used as such to resolve an I: ICalculator interface:

  ```
  var I: ICalculator;
  begin
    if fClient.Services.Info(TypeInfo(ICalculator)).Get(I) then
      ... use I
  ```

**Function** RetrieveSignature: RawUTF8; virtual; abstract;

- Retrieve the published signature of this interface
  - is always available on TServiceFactoryServer, but TServiceFactoryClient will be able to retrieve it only if TServiceContainerServer.PublishSignature is set to TRUE (which is not the default setting, for security reasons)
property Contract: RawUTF8 read fContract;

The service contract, serialized as a JSON object
- a "contract" is in fact the used interface signature, i.e. its implementation mode (InstanceCreation) and all its methods definitions
- a possible value for a one-method interface defined as such:
  function ICalculator.Add(n1,n2: integer): integer;
may be returned as the following JSON object:
{"contract":"Calculator","implementation":"shared",
"methods":[
  {"method":"Add",
   "arguments":[
     {"argument":"Self","direction":"in","type":"self"},
     {"argument":"n1","direction":"in","type":"integer"},
     {"argument":"n2","direction":"in","type":"integer"},
     {"argument":"Result","direction":"out","type":"integer"}
  ]
]}

property ContractExpected: RawUTF8 read fContractExpected write fContractExpected;

The published service contract, as expected by both client and server
- by default, will contain ContractHash property value (for security)
- but you can override this value using plain Contract or any custom value (e.g. a custom version number) - in this case, both TServiceFactoryClient and TServiceFactoryServer instances must have a matching ContractExpected
- this value is returned by a '__contract__' pseudo-method name, with the URI:
  POST /root/Interface.__contract__
  (...)
  {"method":"__contract__","params":[]}
  (e.g. to be checked in TServiceFactoryClient.Create constructor)
- if set to SERVICE_CONTRACT_NONE_EXPECTED (i.e. '*'), the client won't check and ask the server contract for consistency: it may be used e.g. for accessing a plain REST HTTP server which is not based on mORMot, so may not implement POST /root/Interface.__contract__

property ContractHash: RawUTF8 read fContractHash;

A hash of the service contract, serialized as a JSON string
- this may be used instead of the JSON signature, to enhance security (i.e. if you do not want to publish the available methods, but want to check for the proper synchronization of both client and server)
- a possible value may be: "C351335A7406374C"

property InstanceCreation: TServiceInstanceImplementation read fInstanceCreation;

How each class instance is to be created
- only relevant on the server side; on the client side, this class will be accessed only to retrieve a remote access instance, i.e. sicSingle

property InterfaceFactory: TInterfaceFactory read fInterface;

Access to the registered Interface RTTI information

property InterfaceIID: TGUID read GetInterfaceIID;

The registered Interface GUID
- just maps InterfaceFactory.InterfaceIID
property InterfaceMangledURI: RawUTF8 read fInterfaceMangledURI;
    The registered Interface mangled URI
    - in fact this is encoding the GUID using BinToBase64URI(), e.g.
      ['{c9a646d3-9c61-4cb7-bfcd-e12522c8f633}'] into '00amyWGct8_yze41Isj2Mw'
      - can be substituted to the clear InterfaceURI name

property InterfaceTypeInfo: PTypeInfo read GetInterfaceTypeInfo;
    The registered Interface low-level Delphi RTTI type
    - just maps InterfaceFactory.InterfaceTypeInfo

property InterfaceURI: RawUTF8 read fInterfaceURI;
    The registered Interface URI
    - in fact this is the Interface name without the initial 'I', e.g. 'Calculator' for ICalculator

property Rest: TSQLRest read fRest;
    The associated RESTful instance

TServiceFactoryServerInstance = object(TObject)
    Server-side service provider uses this to store one internal instance
    - used by TServiceFactoryServer in sicClientDriven, sicPerSession, sicPerUser or sicPerGroup mode

    Instance: TInterfacedObject;
        The implementation instance itself

    InstanceID: PtrUInt;
        The internal Instance ID, as remotely sent in "id":1
        - is set to 0 when an entry in the array is free

    LastAccess64: Int64;
        GetTickCount64() timestamp corresponding to the last access of this instance

    Session: cardinal;
        The associated client session

procedure SafeFreeInstance(Factory: TServiceFactoryServer);
    Used to release the implementation instance
    - direct FreeAndNil(Instance) may lead to A/V if self has been assigned to an interface to any
      sub-method on the server side -> dec(RefCount)

TServiceFactoryServer = class(TServiceFactory)
    A service provider implemented on the server side
    - each registered interface has its own TServiceFactoryServer instance, available as one
      TSQLServiceContainerServer item from TSQLRest.Services property
      - will handle the implementation class instances of a given interface
        - by default, all methods are allowed to execution: you can call AllowAll, DenyAll, Allow or Deny in
          order to specify your exact security policy

    Used for DI-2.1.5 (page 2547).
constructor Create(aRestServer: TSQLRestServer; aInterface: PTypeInfo;
aInstanceCreation: TServiceInstanceImplementation; aImplementationClass:
TInterfacedClass; const aContractExpected: RawUTF8; aTimeOutSec: cardinal;
aSharedInstance: TInterfacedObject); reintroduce;

Initializes the service provider on the server side
- expect an direct server-side implementation class, which may inherit from plain
TInterfacedClass, TInterfacedObjectWithCustomCreate if you need an overridden constructor, or
TInjectableObject to support DI/IoC
- for sicClientDriven, sicPerSession, sicPerUser or sicPerGroup modes, a time out (in seconds)
can be defined (default is 30 minutes) - if the specified aTimeOutSec is 0, interface will be forced
in sicSingle mode
- you should usually have to call the TSQLRestServer.ServiceRegister() method instead of calling
this constructor directly

destructor Destroy; override;

Releases all used memory
- e.g. any internal TServiceFactoryServerInstance instances (any shared instance, and all still
living instances in sicClientDrive mode)

function Allow(const aMethod: array of RawUTF8): TServiceFactoryServer;
Allow specific methods execution for the all TSQLAuthGroup
- methods names should be specified as an array (e.g. ['Add', 'Multiply'])
- all Groups will be affected by this method (on both client and server sides)
- this method returns self in order to allow direct chaining of security calls, in a fluent interface

function AllowAll: TServiceFactoryServer;
Allow all methods execution for all TSQLAuthGroup
- all Groups will be affected by this method (on both client and server sides)
- this method returns self in order to allow direct chaining of security calls, in a fluent interface

function AllowAllByID(const aGroupID: array of TID): TServiceFactoryServer;
Allow all methods execution for the specified TSQLAuthGroup ID(s)
- the specified group ID(s) will be used to authorize remote service calls from the client side
- you can retrieve a TSQUser ID from its identifier, as such:
  UserGroupID := fServer.MainFieldID(TSQLAuthGroup, 'User');
- this method returns self in order to allow direct chaining of security calls, in a fluent interface

function AllowAllByName(const aGroup: array of RawUTF8): TServiceFactoryServer;
Allow all methods execution for the specified TSQLAuthGroup names
- is just a wrapper around the other AllowAllByID() method, retrieving the Group ID from its
  main field
- this method returns self in order to allow direct chaining of security calls, in a fluent interface

function AllowByID(const aMethod: array of RawUTF8; const aGroupID: array of TID): TServiceFactoryServer;
Allow specific methods execution for the specified TSQLAuthGroup ID(s)
- methods names should be specified as an array (e.g. ['Add', 'Multiply'])
- the specified group ID(s) will be used to authorize remote service calls from the client side
- you can retrieve a TSQUser ID from its identifier, as such:
  UserGroupID := fServer.MainFieldID(TSQLAuthGroup, 'User');
- this method returns self in order to allow direct chaining of security calls, in a fluent interface
function AllowByName(const aMethod: array of RawUTF8; const aGroup: array of RawUTF8): TServiceFactoryServer;

Allow specific methods execution for the specified TSQLAuthGroup name(s)
- is just a wrapper around the other AllowByID() method, retrieving the Group ID from its main field
- methods names should be specified as an array (e.g. ['Add','Multiply'])
- this method returns self in order to allow direct chaining of security calls, in a fluent interface

function Deny(const aMethod: array of RawUTF8): TServiceFactoryServer;

Deny specific methods execution for the all TSQLAuthGroup
- methods names should be specified as an array (e.g. ['Add','Multiply'])
- all Groups will be affected by this method (on both client and server sides)
- this method returns self in order to allow direct chaining of security calls, in a fluent interface

function DenyAll: TServiceFactoryServer;

Deny all methods execution for all TSQLAuthGroup
- all Groups will be affected by this method (on both client and server sides)
- this method returns self in order to allow direct chaining of security calls, in a fluent interface

function DenyAllByID(const aGroupID: array of TID): TServiceFactoryServer;

Deny all methods execution for the specified TSQLAuthGroup ID(s)
- the specified group ID(s) will be used to authorize remote service calls from the client side
- you can retrieve a TSQLAuthGroup ID from its identifier, as such:
  UserGroupID := fServer.MainFieldID(TSQLAuthGroup, 'User');
- this method returns self in order to allow direct chaining of security calls, in a fluent interface

function DenyAllByName(const aGroup: array of RawUTF8): TServiceFactoryServer;

Deny all methods execution for the specified TSQLAuthGroup names
- is just a wrapper around the other DenyAllByID() method, retrieving the Group ID from its main field
- this method returns self in order to allow direct chaining of security calls, in a fluent interface

function DenyByID(const aMethod: array of RawUTF8; const aGroupID: array of TID): TServiceFactoryServer; overload;

Deny specific methods execution for the specified TSQLAuthGroup ID(s)
- methods names should be specified as an array (e.g. ['Add','Multiply'])
- the specified group ID(s) will be used to unauthorize remote service calls from the client side
- you can retrieve a TSQLAuthGroup ID from its identifier, as such:
  UserGroupID := fServer.MainFieldID(TSQLAuthGroup, 'User');
- this method returns self in order to allow direct chaining of security calls, in a fluent interface

function DenyByName(const aMethod: array of RawUTF8; const aGroup: array of RawUTF8): TServiceFactoryServer;

Deny specific methods execution for the specified TSQLAuthGroup name(s)
- is just a wrapper around the other DenyByID() method, retrieving the Group ID from its main field
- methods names should be specified as an array (e.g. ['Add','Multiply'])
- this method returns self in order to allow direct chaining of security calls, in a fluent interface
**function** `Get(out Obj): Boolean; override;`

*Retrieve an instance of this interface from the server side*
- sicShared mode will retrieve the shared instance
- sicPerThread mode will retrieve the instance corresponding to the current running thread
- all other kind of instance creation will behave the same as sicSingle when accessed directly from this method, i.e. from server side: in fact, on the server side, there is no notion of client, session, user nor group
- if ServiceContext.Factory is nil (i.e. if there is no other service context currently associated), this method will also update ServiceContext.Factory, so that the implementation method will be able to access the associated TSQLRestServer instance if needed

**function** `RestServer: TSQLRestServer;`

*Just type-cast the associated TSQLRest instance to a true TSQLRestServer*

**function** `RetrieveSignature: RawUTF8; override;`

*Retrieve the published signature of this interface*
- is always available on TServiceFactoryServer, but TServiceFactoryClient will be able to retrieve it only if TServiceContainerServer.PublishSignature is set to TRUE (which is not the default setting, for security reasons)*

**function** `RunOnAllInstances(const aEvent: TOnServiceFactoryServerOne; var aOpaque): integer;`

*Call the supplied aEvent callback for all class instances implementing this service*

**function** `SetOptions(const aMethod: array of RawUTF8; aOptions: TServiceMethodOptions; aAction: TServiceMethodOptionsAction=moaReplace): TServiceFactoryServer;`

*Define execution options for a given set of methods*
- methods names should be specified as an array (e.g. ['Add','Multiply'])
- if no method name is given (i.e. []), option will be set for all methods
- include optExecInMainThread will force the method(s) to be called within a RunningThread.Synchronize() call - slower, but thread-safe
- this method returns self in order to allow direct chaining of security calls, in a fluent interface

**function** `SetServiceLog(const aMethod: array of RawUTF8; aLogRest: TSQLRest; aLogClass: TSQLRecordServiceLogClass=nil): TServiceFactoryServer;`

*Log method execution information to a TSQLRecordServiceLog table*
- methods names should be specified as an array (e.g. ['Add','Multiply'])
- if no method name is given (i.e. []), option will be set for all methods
- will write to the specified aLogRest instance, and will disable writing if aLogRest is nil
- will write to a (inherited) TSQLRecordServiceLog table, as available in TSQLRest's model, unless a dedicated table is specified as aLogClass
- this method returns self in order to allow direct chaining of security calls, in a fluent interface

**function** `SetTimeoutSec(value: cardinal): TServiceFactoryServer;`

*Define the the instance life time-out, in seconds*
- for sicClientDriven, sicPerSession, sicPerUser or sicPerGroup modes
- raise an exception for other kind of execution
- this method returns self in order to allow direct chaining of setting calls for the service, in a fluent interface
**procedure** AddInterceptor(const Hook: TServiceMethodExecuteEvent);

*Allow to hook the methods execution*
- several events could be registered, and will be called directly before and after method execution
- if optInterceptInputOutput is defined in Options, then Sender.Input/Output fields will contain the execution data context when Hook is called
- see OnMethodExecute if you want to implement security features

**property** ByPassAuthentication: boolean read fByPassAuthentication write fByPassAuthentication;

*Set to TRUE disable Authentication method check for the whole interface*
- by default (FALSE), all interface-based services will require valid RESTful authentication (if enabled on the server side); setting TRUE will disable authentication for all methods of this interface (e.g. for returning some HTML content from a public URI, or to implement a public service catalog)

**property** ExcludeServiceLogCustomAnswer: boolean read fExcludeServiceLogCustomAnswer write fExcludeServiceLogCustomAnswer;

*Disable base64-encoded TSQLRecordServiceLog.Output for methods returning TServiceCustomAnswer record (to reduce storage size)*

**property** ImplementationClass: TInterfacedClass read fImplementationClass;

*The class type used to implement this interface*

**property** OnMethodExecute: TOnServiceCanExecute read fOnMethodExecute write fOnMethodExecute;

*You can define here an event to allow/deny execution of any method of this service, at runtime*

**property** ResultAsJSONObject: boolean read fResultAsJSONObject write fResultAsJSONObject;

*Set to TRUE to return the interface's methods result as JSON object*
- by default (FALSE), any method execution will return a JSON array with all VAR/OUT parameters, in order
- TRUE will generate a JSON object instead, with the VAR/OUT parameter names as field names (and "Result" for any function result) - may be useful e.g. when working with JavaScript clients
- Delphi clients (i.e. TServiceFactoryClient/TInterfacedObjectFake) will transparently handle both formats
- this value can be overridden by setting ForceServiceResultAsJSONObject for a given TSQLRestServerURIContext (e.g. for server-side JavaScript work)

**property** ResultAsJSONObjectWithoutResult: boolean read fResultAsJSONObjectWithoutResult write fResultAsJSONObjectWithoutResult;

*Set to TRUE to return the interface's methods result as JSON object with no '{"result":{...}}' nesting*
- could be used e.g. for plain non mORMot REST Client with in sicSingle or sicShared mode kind of services
- on client side, consider using TSQLRestClientURI.ServiceDefineSharedAPI
property ResultAsXMLObject: boolean read fResultAsXMLObject write fResultAsXMLObject;

  Set to TRUE to return the interface's methods result as XML object
- by default (FALSE), method execution will return a JSON array with all VAR/OUT parameters, or
a JSON object if ResultAsJSONObject is TRUE
- TRUE will generate a XML object instead, with the VAR/OUT parameter names as field names
(and "Result" for any function result) - may be useful e.g. when working with some XML-only
clients
- Delphi clients (i.e. TServiceFactoryClient/TInterfacedObjectFake) does NOT handle this XML format yet
- this value can be overridden by setting ForceServiceResultAsXMLObject for a given
TSQLRestServerURIContext instance

property ResultAsJSONObjectIfAcceptOnlyXML: boolean read fResultAsJSONObjectIfAccept write fResultAsJSONObjectIfAccept;

  Set to TRUE to return XML objects for the interface's methods result if the Accept: HTTP header is exactly 'application/xml' or 'text/xml'
- the header should be exactly 'Accept: application/xml' or 'Accept: text/xml' (and no other value)
- in this case, ForceServiceResultAsXMLObject will be set for this particular
TSQLRestServerURIContext instance, and result returned as XML
- using this method allows to mix standard JSON requests (from JSON or AJAX clients) and XML
requests (from XML-only clients)

property ResultAsXMLObjectNameSpace: RawUTF8 read fResultAsXMLObjectNameSpace write fResultAsXMLObjectNameSpace;

  Specify a custom name space content when returning a XML object
- by default, no name space will be appended - but such rough XML will have potential validation
problems
- you may use e.g. XMLUTF8_NAMESPACE, which will append <content ...> ... </content> around
the generated XML data

property Stat[const aMethod: RawUTF8]: TSynMonitorInputOutput read GetStat;

  Retrieve detailed statistics about a method use
- will return a reference to the actual item in Stats[]: caller should not free the returned instance

property Stats: TSynMonitorInputOutputObjArray read fStats;

  Direct access to per-method detailed process statistics
- this Stats[] array follows Interface.Methods[] order
- see Stat[] property to retrieve information about a method by name

property TimeoutSec: cardinal read GetTimeoutSec write SetTimeoutSecSecInt;

  The instance life time-out, in seconds
- for sicClientDriven, sicPerSession, sicPerUser or sicPerGroup modes
- raise an exception for other kind of execution
- you can also use the SetTimeoutSec() fluent function instead
TServiceFactoryClient = class(TServiceFactory)

A service provider implemented on the client side
- each registered interface has its own TServiceFactoryClient instance, available as one
  TSQLServiceContainerClient item from TSQLRest.Services property
- will emulate "fake" implementation class instance of a given interface and call remotely the
  server to process the actual implementation

Used for DI-2.1.5 (page 2547).

constructor Create(aRest: TSQLRest; aInterface: PTypeInfo; aInstanceCreation: TServiceInstanceImplementation; const aContractExpected: RawUTF8='');

- Initialize the service provider parameters
  - it will check and retrieve all methods of the supplied interface, and prepare all internal
    structures for its serialized execution
  - also set the inherited TServiceInstanceImplementation property
  - initialize fSharedInstance if aInstanceCreation is sicShared
  - it will also ensure that the corresponding TServiceFactory.Contract matches on both client and
    server sides, either by comparing the default signature (based on methods and arguments),
    either by using the supplied expected contract (which may be a custom version number)

destructor Destroy; override;

- Finalize the service provider used instance
  - e.g. the shared fake implementation instance

function Get(out Obj): Boolean; override;

- Retrieve an instance of this interface from the client side

class function GetErrorMessage(status: integer): RawUTF8;

- Convert a HTTP error from mORMot's REST/SA into an English text message
  - will recognize the HTTP_UNAVAILABLE, HTTP_NOTIMPLEMENTED, HTTP_NOTFOUND,
    HTTP_NOTALLOWED, HTTP_UNAUTHORIZED or HTTP_NOTACCEPTABLE errors, as generated by
    the TSQLRestServer side
  - is used by TServiceFactoryClient.InternalInvoke, but may be called on client side for
    TServiceCustomAnswer.Status <> HTTP_SUCCESS

function RetrieveSignature: RawUTF8; override;

- Retrieve the published signature of this interface
  - TServiceFactoryClient will be able to retrieve it only if
    TServiceContainerServer.PublishSignature is set to TRUE (which is not the default setting, for
    security reasons) - this function is always available on TServiceFactoryServer side

function SendNotificationsPending: integer;

- Compute how many pending notifications are waiting for background process initiated by
  SendNotifications() method
procedure SendNotifications(aRest: TSQLRest; aLogClass: TSQLRecordServiceNotificationsClass; aRetryPeriodSeconds: Integer=30; aRemote: TSQLRestClientURI=nil);

Allow background process of method with no results, via a temporary database, to be used e.g. for safe notifications transmission
- will call StoreNotifications() and start background notification
- expect a REST instance, which will store all methods without any results (i.e. procedure without any var/out parameters) on the associated TSQLRecordServiceNotifications class
- a background thread will be used to check for pending notifications, and send them to the supplied aRemote TSQLRestClient instance, or to the main TServiceFactoryClient.fClient instance
- if the remote client is not reachable, will retry after the specified period of time, in seconds
- this method is not blocking, and will write the pending calls to the aRest/aLogClass table, which will be retrieved asynchronously by the background thread

procedure SendNotificationsWait(aTimeOutSeconds: integer);

Wait for all pending notifications to be sent
- you can supply a time out period after which no wait will take place

procedure SetOptions(const aMethod: array of RawUTF8; aOptions: TServiceMethodOptions; aAction: TServiceMethodOptionsAction=moaReplace);

Define execution options for a given set of methods
- methods names should be specified as an array (e.g. ['Add','Multiply'])
- if no method name is given (i.e. []), option will be set for all methods
- only supports optNoLogInput and optNoLogOutput on the client side, by design of "fake" interface remote execution

procedure StoreNotifications(aRest: TSQLRest; aLogClass: TSQLRecordServiceNotificationsClass);

Persist all service calls into a database instead of calling the client
- expect a REST instance, which will store all methods without any results (i.e. procedure without any var/out parameters) on the associated TSQLRecordServiceNotifications class
- once set, regular fClient.URI() won't be called but a new aLogClass entry will be stored in aRest
- to disable this redirection, set aRest and aLogClass to nil

property DelayedInstance: boolean read fDelayedInstance write fDelayedInstance;

Delay the sicClientDriven server-side instance to the first method call
- by default, CreateFakeInstance will call _instance_ server pseudo-method to ensure a fClientDrivenID is safely and properly initialized
- if you are sure that your client's interface variables will be thread-safe, you may define this property to TRUE so that the "id" field as returned at first method call will be used - makes sense only if a lot of short-live interface instances are expected to be generated by the client

property ForcedURI: RawUTF8 read fForcedURI write fForcedURI;

Could be used to force the remote URI to access the service
- by default, the URI will be Root/Calculator or Root/InterfaceMangledURI but you may use this property to use another value, e.g. if you are accessssing a non mORMot REST server (probably with aContractExpected set to SERVICE_CONTRACT_NONE_EXPECTED, and running Client.ServerTimestamp := TimeLogNowUTC to avoid an unsupported ServerTimestampSynchronize call)
property NonBlockWithoutAnswer: boolean read fNonBlockWithoutAnswer write fNonBlockWithoutAnswer;

If methods expecting no result (i.e. plain procedure without var/out parameters) should not block the client waiting for answer
- may be handy e.g. when consuming an event-driven asynchronous service
- will call CallbackNonBlockingSetHeader, currently implemented only in TSQHttpClientWebsockets, with frame gathering

property ParamsAsJSONObject: boolean read fParamsAsJSONObject write fParamsAsJSONObject;

Set to TRUE to send the interface's methods parameters as JSON object
- by default (FALSE), any method execution will send a JSON array with all CONST/VAR parameters, in order
- TRUE will generate a JSON object instead, with the CONST/VAR parameter names as field names - may be useful e.g. when working with a non mORMot server, or when the mORMot server exposes a public API
- defined e.g. by TSQLRestClientURI.ServiceDefineSharedAPI() method

property ResultAsJSONObjectWithoutResult: boolean read fResultAsJSONObjectWithoutResult write fResultAsJSONObjectWithoutResult;

Set to TRUE if the interface's methods result is expected to be a JSON object without the 

TServiceContainerInterface = record
    Used to lookup one service in a global list of interface-based services
    InterfaceName: RawUTF8;
        One 'service' item, as set at URI, e.g. 'Calculator'
    Service: TServiceFactory;
        The associated service provider

TServiceContainerInterfaceMethod = record
    Used to lookup one method in a global list of interface-based services
    InterfaceDotMethodName: RawUTF8;
        One 'service.method' item, as set at URI
        - e.g. 'Calculator.Add','Calculator.Multiply'...
InterfaceMethodIndex: integer;

The index of the method for the given service
- 0..2 indicates _free_/_contract_/signature_pseudo-methods
- then points to InterfaceService.Interface.Methods[InterfaceMethodIndex-3]

InterfaceService: TServiceFactory;
The associated service provider

**TServiceContainer = class(TInterfaceResolverInjected)**

A global services provider class
- used to maintain a list of interfaces implementation
- inherits from TInterfaceResolverInjected and its Resolve() methods, compatible with TInjectableObject

*Used for Di-2.1.5 (page 2547).*

**constructor** Create(aRest: TSQLRest); virtual;
Initialize the list

**destructor** Destroy; override;
Release all registered services

**function** AddInterface(const aInterfaces: array of PTypeInfo; aInstanceCreation: TServiceInstanceImplementation; aContractExpected: RawUTF8=''): boolean; overload;

Method called on the client side to register a service via its interface(s)
- will add a TServiceFactoryClient instance to the internal list
- is called e.g. by TSQLRestClientURI.ServiceRegister or even by TSQLRestServer.ServiceRegister(aClient: TSQLRest...) for a remote access - use TServiceContainerServer.AddImplementation() instead for normal server side implementation
- will raise an exception on error
- will return true if some interfaces have been added
- will check for the availability of the interfaces on the server side, with an optional custom contract to be used instead of methods signature (only for the first interface)

**function** AddInterface(aInterface: PTypeInfo; aInstanceCreation: TServiceInstanceImplementation; const aContractExpected: RawUTF8=''): TServiceFactoryClient; overload;

Method called on the client side to register a service via one interface
- overloaded method returning the corresponding service factory client, or nil on error

**function** AsJson: RawJSON;
Retrieve all registered Services contracts as a JSON array
- i.e. a JSON array of TServiceFactory.Contract JSON objects

**function** CallBackUnRegister(const Callback: IInvoicable): boolean; virtual;
Notify the other side that the given Callback event interface is released
- this default implementation will do nothing

**function** Count: integer;
Return the number of registered service interfaces
function Index(aIndex: integer): TServiceFactory; overload;

*Retrieve a service provider from its index in the list*
- returns nil if out of range index

function Info(const aGUID: TGUID): TServiceFactory; overload;

*Retrieve a service provider from its GUID / Interface type*
- you shall have registered the interface by a previous call to
  TInterfaceFactory_RegisterInterfaces([TypeInfo<IMyInterface>,...])
- on match, it will return the service the corresponding interface factory
- returns nil if the GUID does not match any registered interface
- can be used as such to resolve an I: ICalculator interface
  if fClient.Services.Info(ICalculator).Get(I) then
  ... use I

function Info(aTypeInfo: PTypeInfo): TServiceFactory; overload; virtual;

*Retrieve a service provider from its type information*
- on match, it will return the service the corresponding interface factory
- returns nil if the type information does not match any registered interface
- can be used as such to resolve an I: ICalculator interface
  if fClient.Services.Info(TypeInfo(ICalculator)).Get(I) then
  ... use I

- is defined as virtual so that e.g. TServiceContainerClient will automatically register the
  interface, if it was not already done

procedure Release;

*Release all services of a TSQLRest instance before shutdown*
- will allow to properly release any pending callbacks
- TSQLRest.Services.Release will call FreeAndNil(fServices)

procedure SetGUIDs(out Services: TGUIDDynArray);

*Retrieve all registered Services TGUID*

procedure SetInterfaceNames(out Names: TRawUTF8DynArray);

*Retrieve all registered Services names*
- i.e. all interface names without the initial 'I', e.g. 'Calculator' for ICalculator

property ExpectMangledURI: boolean read fExpectMangledURI write SetExpectMangledURI;

*Set if the URI is expected to be mangled from the GUID*
- by default (FALSE), the clear service name is expected to be supplied at the URI level (e.g. 'Calculator')
- if this property is set to TRUE, the mangled URI value will be expected instead (may enhance security) - e.g. '00amyWGct0y_ze4Isj2Mw'

property Rest: TSQLRest read fRest;

*The associated RESTful instance*
### property Services[const aURI: RawUTF8]: TServiceFactory read GetService;

Retrieve a service provider from its URI
- it expects the supplied URI variable to be e.g. '00amyWGct0y_ze4ljs2Mw' or 'Calculator', depending on the ExpectMangledURI property
- on match, it will return the service the corresponding interface factory
- returns nil if the URI does not match any registered interface

### property ServicesFactoryClients: TServiceFactoryClientClass read fServicesFactoryClients; write fServicesFactoryClients;

The services factory client classes
- by default, will use TServiceFactoryClient

---

**IServiceRecordVersionCallback** = interface(IInvokable)

A callback interface used to notify a TSQLRecord modification in real time
- will be used e.g. by TSQLRestServer.RecordVersionSynchronizeSubscribeMaster()
- all methods of this interface will be called asynchronously when transmitted via our WebSockets implementation, since they are defined as plain procedures
- each callback instance should be private to a specific TSQLRecord

**procedure** Added(const NewContent: RawJSON);

This event will be raised on any Add on a versioned record
- the supplied JSON object will contain the TRecordVersion field

**procedure** CurrentFrame(isLast: boolean);

Allow to optimize process for WebSockets "jumbo frame" items
- this method may be called with isLast=false before the first method call of this interface, then with isLast=true after the call of the last method of the "jumbo frame"
- match TInterfaceFactory.MethodIndexCurrentFrameCallback signature
- allow e.g. to create a temporary TSQLRestBatch for jumbo frames
- if individual frames are received, this method won't be called

**procedure** Deleted(const ID: TID; const Revision: TRecordVersion);

This event will be raised on any Delete on a versioned record

**procedure** Updated(const ModifiedContent: RawJSON);

This event will be raised on any Update on a versioned record
- the supplied JSON object will contain the TRecordVersion field

---

**IServiceRecordVersion** = interface(IInvokable)

Service definition for master/slave replication notifications subscribe
- implemented by TServiceRecordVersion, as used by TSQLRestServer.RecordVersionSynchronizeMasterStart(), and expected by TSQLRestServer.RecordVersionSynchronizeSlaveStart()

**function** Subscribe(const SQLTableName: RawUTF8; const revision: TRecordVersion; const callback: IServiceRecordVersionCallback): boolean;

Will register the supplied callback for the given table
IServiceWithCallbackReleased = interface(IInvokable)

Service definition with a method which will be called when a callback interface instance is released on the client side
- may be used to implement safe publish/subscribe mechanism using interface callbacks, e.g. over WebSockets

procedure CallbackReleased(const callback: IInvokable; const interfaceName: RawUTF8);

Will be called when a callback is released on the client side
- this method matches the TInterfaceFactory.MethodIndexCallbackReleased signature, so that it will be called with the interface instance by TServiceContainerServer.FakeCallbackRelease
- you may use it as such - see sample Project31ChatServer.dpr:

procedure TChatService.CallbackReleased(const callback: IInvokable; const interfaceName: RawUTF8);
begin
  // unsubscribe from fConnected: array of IChatCallback
  if interfaceName='IChatCallback' then
    InterfaceArrayDelete(fConnected, callback);
end;

TServiceContainerServer = class(TServiceContainer)

A services provider class to be used on the server side
- this will maintain a list of true implementation classes

Used for DI-2.1.5 (page 2547).

constructor Create(aRest: TSQLRest); override;
  Initialize the list

destructor Destroy; override;
  Finalize the service container

function AddImplementation(aImplementationClass: TInterfacedClass; const aInterfaces: array of PTypeInfo; aInstanceCreation: TServiceInstanceImplementation; aSharedImplementation: TInterfacedObject; const aContractExpected: RawUTF8): TServiceFactoryServer;

Method called on the server side to register a service via its interface(s) and a specified implementation class or a shared instance (for sicShared mode)
- will add a TServiceFactoryServer instance to the internal list
- will raise an exception on error
- will return the first of the registered TServiceFactoryServer created on success (i.e. the one corresponding to the first item of the aInterfaces array), or nil if registration failed (e.g. if any of the supplied interfaces is not implemented by the given class)
- the same implementation class can be used to handle several interfaces (just as Delphi allows to do natively)
class function CallbackReleasedOnClientSide(const callback: IInterface; callbackText: PShortString=nil): boolean; overload;

Class method able to check if a given server-side callback event fake instance has been released on the client side
- may be used to automatically purge a list of subscribed callbacks, e.g. before triggering the interface instance, and avoid an exception
- can optionally append the callback class instance information to a local shortstring variable, e.g. for logging/debug purposes

function RecordVersionSynchronizeSubscribeMaster(TableIndex: integer; RecordVersion: TRecordVersion; const SlaveCallback: TServiceRecordVersionCallback): boolean;

Register a callback interface which will be called each time a write operation is performed on a given TSQlRecord with a TRecordVersion field
- called e.g. by TSQLRestServer.RecordVersionSynchronizeSubscribeMaster

procedure RecordVersionNotifyAddUpdate(Occasion: TSQLOccasion; TableIndex: integer; const Decoder: TJSONObjectDecoder); overload;

Notify any TRecordVersion callback for a table Add/Update from a TJSONObjectDecoder content
- used e.g. by TSQLRestStorageMongoDB.DocFromJSON()

procedure RecordVersionNotifyAddUpdate(Occasion: TSQLOccasion; TableIndex: integer; const Document: TDocVariantData); overload;

Notify any TRecordVersion callback for a table Add/Update from a TDocVariant content
- used e.g. by TSQLRestStorageMongoDB.DocFromJSON()

procedure RecordVersionNotifyDelete(TableIndex: integer; const ID: TID; const Revision: TRecordVersion);

Notify any TRecordVersion callback for a table Delete

procedure SetServiceLog(aLogRest: TSQLRest; aLogClass: TSQLRecordServiceLogClass=nil; const aExcludedMethodNamesCSV: RawUTF8='');

Log method execution information to a TSQLRecordServiceLog table
- TServiceFactoryServer.SetServiceLog() will be called for all registered interfaced-based services of this container
- will write to the specified aLogRest instance, and will disable writing if aLogRest is nil
- will write to a (inherited) TSQLRecordServiceLog table, as available in TSQLRest's model, unless a dedicated table is specified as aLogClass
- you could specify a CSV list of method names to be excluded from logging (containing e.g. a password or a credit card number), containing either the interface name (as 'ICalculator.Add'), or not (as 'Add')

property CallbackOptions: TServiceCallbackOptions read fCallbackOptions write fCallbackOptions;

Defines how SOA callbacks will be handled
property OnCallbackReleasedOnClientSide: TOnCallbackReleased read fOnCallbackReleasedOnClientSide;

This event will be launched when a callback interface is notified as released on the Client side
- as an alternative, you may define the following method on the registration service interface type, which will be called when a callback registered via this service is released (e.g. to unsubscribe the callback from an interface list, via InterfaceArrayDelete):

```
procedure CallbackReleased(const callback: IInvokable; const interfaceName: RawUTF8);
```

property OnCallbackReleasedOnServerSide: TOnCallbackReleased read fOnCallbackReleasedOnServerSide;

This event will be launched when a callback interface is released on the Server side

property PublishSignature: boolean read fPublishSignature write fPublishSignature;

Defines if the "method":"_signature_" or /root/Interface._signature pseudo method is available to retrieve the whole interface signature, encoded as a JSON object
- is set to FALSE by default, for security reasons: only "_contract_" pseudo method is available - see TServiceContainer.ContractExpected

property SessionTimeout: cardinal read fSessionTimeout write fSessionTimeout;

The default TServiceFactoryServer.TimeoutSec value
- default is 30 minutes
- you can customize each service using its corresponding TimeoutSec property

**TServiceRecordVersion** = class(TInjectableObjectRest)

This class implements a service, which may be called to push notifications for master/slave replication
- as used by TSQLRestServer.RecordVersionSynchronizeMasterStart(), and expected by TSQLRestServer.RecordVersionSynchronizeSlaveStart()

```
function Subscribe(const SQLTableName: RawUTF8; const revision: TRecordVersion; const callback: IServiceRecordVersionCallback): boolean;
```

Will register the supplied callback for the given table

**TServiceContainerClient** = class(TServiceContainer)

A services provider class to be used on the client side
- this will maintain a list of fake implementation classes, which will remotely call the server to make the actual process

Used for DI-2.1.5 (page 2547).

```
function CallBackUnRegister(const Callback: IInvokable): boolean; override;
```

Notify the other side that the given Callback event interface is released
- this overridden implementation will check the private fFakeCallbacks list

```
function Info(aTypeInfo: PTypeInfo): TServiceFactory; overload; override;
```

Retrieve a service provider from its type information
- this overridden method will register the interface, if was not yet made
- in this case, the interface will be registered with sicClientDriven implementation method, unless DisableAutoRegisterAsClientDriven is TRUE
**property** DisableAutoRegisterAsClientDriven: boolean read fDisableAutoRegisterAsClientDriven write fDisableAutoRegisterAsClientDriven;

Allow to disable the automatic registration as sicClientDriven in Info()

TInterfacedCallback = class(TInterfacedObjectLocked)

TInterfacedObject class which will notify a REST server when it is released
- could be used when implementing event callbacks as interfaces, so that the other side instance will be notified when it is destroyed

constructor Create(aRest: TSQLRest; const aGUID: TGUID); reintroduce;

Initialize the instance for a given REST and callback interface

destructor Destroy; override;

Finalize the instance, and notify the TSQLRestServer that the callback is now unreachable
- i.e. will call CallbackRestUnregister

procedure CallbackRestUnregister; virtual;

Notify the associated TSQLRestServer that the callback is disconnected
- i.e. will call TSQLRestServer's TServiceContainer.CallbackUnRegister()
- this method will process the unsubscription only once

property Rest: TSQLRest read fRest;

The associated TSQLRestServer instance, which will be notified when the callback is released

property RestInterface: TGUID read fInterface write fInterface;

The interface type, implemented by this callback class

TBlockingCallback = class(TInterfacedCallback)

Asynchronous callback to emulate a synchronous/blocking process
- once created, process will block via a WaitFor call, which will be released when
CallbackFinished() is called by the process background thread

constructor Create(aTimeOutMs: integer; aRest: TSQLRest; const aGUID: TGUID);

reintroduce;

Initialize the callback instance
- specify a time out milliseconds period after which blocking execution should be handled as
failure (if 0 is set, default 3000 will be used)
- you can optionally set a REST and callback interface for automatic notification when this
TInterfacedCallback will be released

destructor Destroy; override;

Finalize the callback instance

function Reset: boolean; virtual;

Just a wrapper to reset the internal Event state to evNone
- may be used to re-use the same TBlockingCallback instance, after a successfull
WaitFor/CallbackFinished process
- returns TRUE on success (i.e. status was not beWaiting)
- if there is a WaitFor currently in progress, returns FALSE
function WaitFor: TBlockingEvent; virtual;
   Called to wait for the callback to be processed, or trigger timeout
   - will block until CallbackFinished() is called by the processing thread
   - returns the final state of the process, i.e. beRaised or beTimeOut

procedure CallbackFinished(aRestForLog: TSQLRest; aServerUnregister: boolean=false); virtual;
   Should be called by the callback when the process is finished
   - the caller will then let its WaitFor method return
   - if aServerUnregister is TRUE, will also call CallbackRestUnregister to notify the server that the
     callback is no longer needed
   - will optionally log all published properties values to the log class of the supplied REST instance

property Event: TBlockingEvent read GetEvent;
   The current state of process
   - just a wrapper around Process.Event
   - use Reset method to re-use this instance after a WaitFor process

property Process: TBlockingProcess read fProcess;
   The associated blocking process instance

TServiceRecordVersionCallback = class(TInterfacedCallback)
   This class implements a callback interface, able to write all remote ORM notifications to the local
   DB
   - could be supplied as callback parameter, possibly via WebSockets transmission, to
     TSQLRestServer.RecordVersionSynchronizeSubscribeMaster()

constructor Create(aSlave: TSQLRestServer; aMaster: TSQLRestClientURI; aTable: TSQLRecordClass; aOnNotify: TOnBatchWrite); reintroduce;
   Initialize the instance able to apply callbacks for a given table on a local slave REST server from a
   remote master REST server
   - the optional low-level aOnNotify callback will be triggered for each incoming notification, to
     track the object changes in real-time

destructor Destroy; override;
   Finalize this callback instance

procedure Added(const NewContent: RawJSON); virtual;
   This event will be raised on any Add on a versioned record

procedure CurrentFrame(isLast: boolean); virtual;
   Match TInterfaceFactory.MethodIndexCurrentFrameCallback signature, so that
   TSQLHttpClientWebsockets.CallbackRequest will call it
   - it will create a temporary TSQLRestBatch for the whole "jumbo frame"

procedure Deleted(const ID: TID; const Revision: TRecordVersion); virtual;
   This event will be raised on any Delete on a versioned record

procedure Updated(const ModifiedContent: RawJSON); virtual;
   This event will be raised on any Update on a versioned record
property OnNotify: TOnBatchWrite read fOnNotify write fOnNotify;

Low-level event handler triggered by Added/Updated/Deleted methods

IMultiCallbackRedirect = interface(IInterface)

Prototype of a class implementing redirection of a given interface
- as returned e.g. by TSQLRest.MultiRedirect method
- can be used as a main callback, then call Redirect() to manage an internal list of redirections
- when you release this instance, will call Rest.Service.CallbackUnregister with the associated fake callback generated

procedure Redirect(const aCallback: TInterfacedObject; const aMethodsNames: array of RawUTF8; aSubscribe: boolean=true); overload;

Add or remove a class instance callback to the internal redirection list
- will register a callback if aSubscribe is true
- will unregister a callback if aSubscribe is false
- supplied aCallback instance should implement the expected interface GUID
- this method will be implemented as thread-safe
- you can specify some method names, or all methods redirection if []

procedure Redirect(const aCallback: IInvokable; const aMethodsNames: array of RawUTF8; aSubscribe: boolean=true); overload;

Add or remove an interface callback to the internal redirection list
- will register a callback if aSubscribe is true
- will unregister a callback if aSubscribe is false
- supplied aCallback should implement the expected interface GUID
- this method will be implemented as thread-safe
- you can specify some method names, or all methods redirection if []

TSQLRestCacheEntryValue = packed record
For TSQLRestCache, stores a table values

ID: TID;
Corresponding ID

JSON: RawUTF8;
JSON encoded UTF-8 serialization of the record

Tag: cardinal;
Some associated unsigned integer value
- not used by TSQLRestCache, but available at TSQLRestCacheEntry level

Timestamp512: cardinal;
GetTickCount64 shr 9 timestamp when this cached value was stored
- resulting time period has therefore a resolution of 512 ms, and overflows after 70 years without computer reboot
- equals 0 when there is no JSON value cached
TSQLRestCacheEntry = object(TObject)
  \textit{For TSQLRestCache, stores a table settings and values}

  \begin{itemize}
  \item CacheAll: boolean;
    \textit{The whole specified Table content will be cached}
  \item CacheEnable: boolean;
    \textit{TRUE if this table should use caching
    - i.e. if was not set, or worth it for this table (e.g. in-memory table)}
  \item Count: integer;
    \textit{The number of entries stored in Values[]}
  \item Mutex: TSynLocker;
    \textit{Used to lock the table cache for multi thread safety}
  \item TimeOutMS: Cardinal;
    \textit{Time out value (in ms)
    - if 0, caching will never expire}
  \item Value: TDynArray;
    \textit{TDynArray wrapper around the Values[] array}
  \item Values: TSQLRestCacheEntryValueDynArray;
    \textit{All cached IDs and JSON content}
  \end{itemize}

  \textbf{function} CachedMemory(FlushedEntriesCount: PInteger=nil): cardinal;
  Compute how much memory stored entries are using
  - will also flush outdated entries

  \textbf{function} RetrieveJSON(aID: TID; aValue: TSQLRecord; aTag: PCardinal=nil): boolean;
  \textit{Unserialize a JSON cached record of a given ID}

  \textbf{function} RetrieveJSON(aID: TID; var aJSON: RawUTF8; aTag: PCardinal=nil): boolean;
  \textit{Retrieve a JSON serialization of a given ID from cache}

  \textbf{procedure} Clear;
  \textit{Reset all settings corresponding to this table cache}

  \textbf{procedure} Done;
  \textit{Finalize this table cache entry}

  \textbf{procedure} FlushCacheAllEntries;
  \textit{Flush cache for all Value[]}

  \textbf{procedure} FlushCacheEntry(Index: Integer);
  \textit{Flush cache for a given Value[] index}

  \textbf{procedure} Init;
  \textit{Initialize this table cache}
  - will set Value wrapper and Mutex handle - other fields should have been cleared by caller (is the case for a TSQLRestCacheEntryDynArray)
```pascal
procedure SetCache(aID: TID);
Add the supplied ID to the Value[] array

procedure SetJSON(aID: TID; const aJSON: RawUTF8; aTag: cardinal=0); overload;
Update/refresh the cached JSON serialization of a given ID

procedure SetJSON(aRecord: TSQLRecord); overload;
Update/refresh the cached JSON serialization of a supplied Record

TSQLRestCache = class(TObject)
Implement a fast TSQLRecord cache, per ID, at the TSQLRest level
- purpose of this caching mechanism is to speed up retrieval of some common values at either
  Client or Server level (like configuration settings)
- only caching synchronization is about the following RESTful basic commands: RETRIEVE, ADD,
  DELETION and UPDATE (that is, a complex direct SQL UPDATE or via TSQLRecordMany pattern
  won’t be taken into account)
- only Simple fields are cached: e.g. the BLOB fields are not stored
- this cache is thread-safe (access is locked per table)
- this caching will be located at the TSQLRest level, that is no automated synchronization is
  implemented between TSQLRestClient and TSQLRestServer: you shall ensure that your code won’t
  fail due to this restriction

constructor Create(aRest: TSQLRest); reintroduce;
Create a cache instance
- the associated TSQLModel will be used internally

destructor Destroy; override;
Release the cache instance

function CachedEntries: cardinal;
Returns the number of JSON serialization records within this cache

function CachedMemory(FlushedEntriesCount: PInteger=nil): cardinal;
Returns the memory used by JSON serialization records within this cache
- this method will also flush any outdated entries in the cache

function FillFromQuery(aTable: TSQLRecordClass; const FormatSQLWhere: RawUTF8;
const BoundsSQLWhere: array of const): integer;
- will fill the internal JSON cache of a given Table with data coming from a REST query
- returns the number of TSQLRecord items actually cached
- may be handy to pre-load a set of values (e.g. a lookup table) from a single REST query, without
  waiting for each record to be retrieved

function IsCached(aTable: TSQLRecordClass): boolean;
Returns TRUE if the table is part of the current caching policy

function SetCache(aTable: TSQLRecordClass; aID: TID): boolean; overload;
Activate the internal caching for a given TSQLRecord
- if this item is already cached, do nothing
- return true on success
```
function SetCache(aRecord: TSQLRecord): boolean; overload;
   Activate the internal caching for a given TSQLRecord
   - will cache the specified aRecord.ID item
   - if this item is already cached, do nothing
   - return true on success

function SetCache(aTable: TSQLRecordClass; const aIDs: array of TID): boolean; overload;
   Activate the internal caching for a set of specified TSQLRecord
   - if these items are already cached, do nothing
   - return true on success

function SetCache(aTable: TSQLRecordClass): boolean; overload;
   Activate the internal caching for a whole Table
   - any cached item of this table will be flushed
   - return true on success

function SetTimeOut(aTable: TSQLRecordClass; aTimeoutMS: cardinal): boolean;
   Set the internal caching time out delay (in ms) for a given table
   - actual resolution is 512 ms
   - time out setting is common to all items of the table
   - if aTimeout is left to its default 0 value, caching will never expire
   - return true on success

procedure Clear;
   Flush the cache, and destroy all settings
   - this will flush all stored JSON content, AND destroy the settings (SetCache/SetTimeOut) to default (i.e. no cache enabled)

procedure Flush(aTable: TSQLRecordClass); overload;
   Flush the cache for a given table
   - this will flush all stored JSON content, but keep the settings (SetCache/SetTimeOut) as before for this table

procedure Flush; overload;
   Flush the cache
   - this will flush all stored JSON content, but keep the settings (SetCache/SetTimeOut) as before

procedure Flush(aTable: TSQLRecordClass; const aIDs: array of TID); overload;
   Flush the cache for a set of specified records
   - this will flush the stored JSON content for these record (and table settings will be kept)

procedure Flush(aTable: TSQLRecordClass; aID: TID); overload;
   Flush the cache for a given record
   - this will flush the stored JSON content for this record (and table settings will be kept)

procedure Notify(aRecord: TSQLRecord; aAction: TSQLOccasion); overload;
   TSQLRest instance shall call this method when a record is added or updated
   - this overloaded method will call the other Trace method, serializing the supplied aRecord content as JSON (not in the case of seDelete)
procedure Notify(aTableIndex: integer; aID: TID; const aJSON: RawUTF8; aAction: TSQLOccasion); overload;
    TSQRest instance shall call this method when a record is retrieved, added or updated
    - this overloaded method expects the content to be specified as JSON object, and
    - TSQLRecordClass to be specified as its index in Rest.Model.Tables[]

procedure Notify(aTable: TSQLRecordClass; aID: TID; const aJSON: RawUTF8; aAction: TSQLOccasion); overload;
    TSQRest instance shall call this method when a record is added or updated
    - this overloaded method expects the content to be specified as JSON object

procedure NotifyDeletion(aTableIndex: integer; aID: TID); overload;
    TSQRest instance shall call this method when a record is deleted
    - this method is dedicated for a record deletion
    - TSQLRecordClass to be specified as its index in Rest.Model.Tables[]

procedure NotifyDeletion(aTable: TSQLRecordClass; aID: TID); overload;
    TSQRest instance shall call this method when a record is deleted
    - this method is dedicated for a record deletion

procedure NotifyDeletions(aTableIndex: integer; const aIDs: array of Int64); overload;
    TSQRest instance shall call this method when records are deleted
    - TSQLRecordClass to be specified as its index in Rest.Model.Tables[]

property Rest: TSQRest read fRest;
    Read-only access to the associated TSQRest instance

TSQLRestAcquireExecution = class(TSynPersistentLock)
    Used to store the execution parameters for a TSQRest instance
    LockedTimeOut: cardinal;
        Delay before failing to acquire the lock
    Mode: TSQRestServerAcquireMode;
        How read or write operations will be executed
    Thread: TSynBackgroundThreadMethod;
        Background thread instance (if any)

destructor Destroy; override;
    Finalize the memory structure, and the associated background thread

TSQLRestBackgroundTimer = class(TSynBackgroundTimer)
    Thread able to run one or several tasks at a periodic pace, or do asynchronous interface or batch execution, with proper TSQRest integration
    - used e.g. by TSQRest.TimerEnable/AsynchRedirect/AsynchBatchStart methods
    - TSQRest.BackgroundTimer will define one instance, but you may create other dedicated instances to instantiate separated threads
constructor Create(aRest: TSQLRest; const aThreadName: RawUTF8=''; aStats: TSynMonitorClass=nil); reintroduce; virtual;
   Initialize the thread for a periodic task processing

destructor Destroy; override;
   Finalize the thread

function AsynchBatchAdd(Value: TSQLRecord; SendData: boolean; ForceID: boolean=false; const CustomFields: TSQLFieldBits=[]; DoNotAutoComputeFields: boolean=false): integer;
   Create a new ORM member in a BATCH to be written in a background thread
   - should have been preceded by a call to AsynchBatchStart(), or returns -1
   - is a wrapper around TSQLRestBatch.Add() sent in the Timer thread, so will return the index in the BATCH rows, not the created TID
   - this method is thread-safe

function AsynchBatchDelete(Table: TSQLRecordClass; ID: TID): integer;
   Delete an ORM member in a BATCH to be written in a background thread
   - should have been preceded by a call to AsynchBatchStart(), or returns -1
   - is a wrapper around the TSQLRestBatch.Delete() sent in the Timer thread
   - this method is thread-safe

function AsynchBatchRawAdd(Table: TSQLRecordClass; const SentData: RawUTF8): integer;
   Append some JSON content in a BATCH to be written in a background thread
   - could be used to emulate AsynchBatchAdd() with an already pre-computed JSON object
   - is a wrapper around TSQLRestBatch.RawAdd() sent in the Timer thread, so will return the index in the BATCH rows, not the created TID
   - this method is thread-safe

function AsynchBatchStart(Table: TSQLRecordClass; SendSeconds: integer; PendingRowThreshold: integer=500; AutomaticTransactionPerRow: integer=1000; Options: TSQLRestBatchOptions=[boExtendedJSON]): boolean;
   Prepare an asynchronous ORM BATCH process, executed in a background thread
   - will initialize a TSQLRestBatch and call TimerEnable to initialize the background thread, following the given processing period (in seconds), or the TSQLRestBatch.Count threshold to call BatchSend
   - actual REST/CRUD commands will take place via AsynchBatchAdd, AsynchBatchUpdate and AsynchBatchDelete methods
   - only a single AsynchBatch() call per Table is allowed at a time, unless AsynchBatchStop method is used to flush the current asynchronous BATCH
   - using a BATCH in a dedicated thread will allow very fast background asynchronous process of ORM methods, sufficient for most use cases

function AsynchBatchStop(Table: TSQLRecordClass): boolean;
   Finalize asynchronous ORM BATCH process, executed in a background thread
   - should have been preceded by a call to AsynchBatch(), or returns false
   - Table=nil will release all existing batch instances
function AsynchBatchUpdate(Value: TSQLRecord; const CustomFields: TSQLFieldBits=[]; DoNotAutoComputeFields: boolean=false): integer;

*Update an ORM member in a BATCH to be written in a background thread*
- should have been preceded by a call to AsynchBatchStart(), or returns -1
- is a wrapper around the TSQLRestBatch.Update() sent in the Timer thread
- this method is thread-safe

procedure AsynchBatchRawAppend(Table: TSQLRecordClass; SentData: TTextWriter);

*Append some JSON content in a BATCH to be written in a background thread*
- could be used to emulate AsynchBatchAdd() with an already pre-computed JSON object, as stored in a TTextWriter instance
- is a wrapper around TSQLRestBatch.RawAppend.AddNoJSONEscape(SentData) in the Timer thread
- this method is thread-safe

procedure AsynchInterning(Interning: TRawUTF8Interning; InterningMaxRefCount: integer=2; PeriodMinutes: integer=5);

*Allows background garbage collection of specified RawUTF8 interning*
- will run Interning.Clean(2) every 5 minutes by default
- set InterningMaxRefCount=0 to disable process of the Interning instance

procedure AsynchRedirect(const aGUID: TGUID; const aDestinationInterface: IInvokable; out aCallbackInterface; const aOnResult: TOnAsynchRedirectResult=nil);

*Define asynchronous execution of interface methods in a background thread*
- this method implements any interface via a fake class, which will redirect all methods calls into calls of another interface, but as a FIFO in a background thread, shared with TimerEnable/TimerDisable process
- parameters will be serialized and stored as JSON in the queue
- by design, only procedure methods without any output parameters are allowed, since their execution will take place asynchronously
- of course, a slight delay is introduced in aDestinationInterface methods execution, but the main process thread is not delayed any more, and is free from potential race conditions
- the returned fake aCallbackInterface should be freed before TSQLRest is destroyed, to release the redirection resources
- it is an elegant resolution to the most difficult implementation problem of SOA callbacks, which is to avoid race condition on reentrance, e.g. if a callback is run from a thread, and then the callback code try to execute something in the context of the initial thread, protected by a critical section (mutex)
procedure AsynchRedirect(const aGUID: TGUID; const aDestinationInstance: TInterfacedObject; out aCallbackInterface; const aOnResult: TOnAsynchRedirectResult=nil); overload;

Define asynchronous execution of interface methods in a background thread
- this method implements any interface via a fake class, which will redirect all methods calls into calls of another interface, but as a FIFO in a background thread, shared with TimerEnable/TimerDisable process
- parameters will be serialized and stored as JSON in the queue
- by design, only procedure methods without any output parameters are allowed, since their execution will take place asynchronously
- of course, a slight delay is introduced in aDestinationInterface methods execution, but the main process thread is not delayed any more, and is free from potential race conditions
- the returned fake aCallbackInterface should be freed before TSQLRest is destroyed, to release the redirection resources
- it is an elegant resolution to the most difficult implementation problem of SOA callbacks, which is to avoid race condition on reentrance, e.g. if a callback is run from a thread, and then the callback code try to execute something in the context of the initial thread, protected by a critical section (mutex)

property Name: RawUTF8 read fThreadName;

The identifier of the thread, as logged

TSQLRest = class(TObject)
A generic REpresentational State Transfer (REST) client/server class
Used for DI-2.1.1 (page 2543), DI-2.1.1.1 (page 2543).

QueryCustom: array of TSQLQueryCustom;
The custom queries parameters for User Interface Query action

constructor Create(aModel: TSQLModel); virtual;
Initialize the class, and associate it to a specified database Model

destructor Destroy; override;
Release internal used instances
- e.g. release associated TSQLModel or TServiceContainer
function Add(Value: TSQLRecord; SendData: boolean; ForceID: boolean=false; DoNotAutoComputeFields: boolean=false): TID; overload;

Create a new member
- implements REST POST collection
- if SendData is true, client sends the current content of Value with the request, otherwise record is created with default values
- if ForceID is true, client sends the Value.ID field to use this ID for adding the record (instead of a database-generated ID)
- on success, returns the new RowID value; on error, returns 0
- on success, Value.ID is updated with the new RowID
- the TSQLRawBlob(BLOB) fields values are not set by this method, to preserve bandwidth - see UpdateBlobFields() and AddWithBlobs() methods
- the TSQLRecordMany fields are not set either: they are separate instances created by TSQLRecordMany.Create, with dedicated methods to access to the separated pivot table
- this method will call EngineAdd() to perform the request

function Add(Value: TSQLRecord; const CustomCSVFields: RawUTF8; ForceID: boolean=false; DoNotAutoComputeFields: boolean=false): TID; overload;

Create a new member, including selected fields
- implements REST POST collection
- if ForceID is true, client sends the Value.ID field to use this ID for adding the record (instead of a database-generated ID)
- this method will call EngineAdd() to perform the request

function Add(Value: TSQLRecord; const CustomFields: TSQLFieldBits; ForceID: boolean=false; DoNotAutoComputeFields: boolean=false): TID; overload;

Create a new member, including selected fields
- implements REST POST collection
- if ForceID is true, client sends the Value.ID field to use this ID for adding the record (instead of a database-generated ID)
- this method will call EngineAdd() to perform the request

function AddOrUpdate(Value: TSQLRecord; ForceID: boolean=false): TID;

Create or update a member, depending if the Value has already an ID
- implements REST POST if Value.ID=0 or ForceID is set, or a REST PUT collection to update the record pointed by a Value.ID<>0
- will return the created or updated ID

function AddSimple(aTable: TSQLRecordClass; const aSimpleFields: array of const; ForcedID: TID=0): TID;

Create a new member, from a supplied list of field values
- implements REST POST collection
- the aSimpleFields parameters must follow explicitely the order of published properties of the supplied aTable class, excepting the TSQLRawBlob and TSQLRecordMany kind (i.e. only so called "simple fields")
- the aSimpleFields must have exactly the same count of parameters as there are "simple fields" in the published properties
- if ForcedID is set to non null, client sends this ID to be used when adding the record (instead of a database-generated ID)
- on success, returns the new RowID value; on error, returns 0
- call interanlly the Add virtual method above
**function AddWithBlobs**

```
function AddWithBlobs(Value: TSQLRecord; ForceID: boolean=false; DoNotAutoComputeFields: boolean=false): TID; virtual;
```

*Create a new member, including its BLOB fields*
- implements REST POST collection
- this method will create a JSON representation of the document including the BLOB fields as Base64 encoded text, so will be less efficient than a dual Add() + UpdateBlobFields() methods if the binary content has a non trivial size
- this method will call EngineAdd() to perform the request

**function AsynchBatchAdd**

```
function AsynchBatchAdd(Value: TSQLRecord; SendData: boolean; ForceID: boolean=false; const CustomFields: TSQLFieldBits=[]; DoNotAutoComputeFields: boolean=false): integer;
```

*Create a new ORM member in a BATCH to be written in a background thread*
- should have been preceded by a call to AsynchBatchStart(), or returns -1
- is a wrapper around BackgroundTimer.AsynchBatchAdd(), so will return the index in the BATCH rows, not the created TID
- this method is thread-safe

**function AsynchBatchDelete**

```
function AsynchBatchDelete(Table: TSQLRecordClass; ID: TID): integer;
```

*Delete an ORM member in a BATCH to be written in a background thread*
- should have been preceded by a call to AsynchBatchStart(), or returns -1
- is a wrapper around the TSQLRestBatch.Delete() sent in the Timer thread
- this method is thread-safe

**function AsynchBatchRawAdd**

```
function AsynchBatchRawAdd(Table: TSQLRecordClass; const SentData: RawUTF8): integer;
```

*Append some JSON content in a BATCH to be written in a background thread*
- could be used to emulate AsynchBatchAdd() with an already pre-computed JSON object
- is a wrapper around BackgroundTimer.AsynchBatchRawAdd(), so will return the index in the BATCH rows, not the created TID
- this method is thread-safe

**function AsynchBatchStart**

```
function AsynchBatchStart(Table: TSQLRecordClass; SendSeconds: integer; PendingRowThreshold: integer=500; AutomaticTransactionPerRow: integer=1000; Options: TSQLRestBatchOptions=[boExtendedJSON]): boolean;
```

*Prepare an asynchronous ORM BATCH process, executed in a background thread*
- will initialize a TSQLRestBatch and call TimerEnable to initialize the background thread, following the given processing period (in seconds), or the TSQLRestBatch.Count threshold to call BatchSend
- actual REST/CRUD commands will take place via AsynchBatchAdd, AsynchBatchUpdate and AsynchBatchDelete methods
- only a single AsynchBatch() call per Table is allowed at a time, unless AsynchBatchStop method is used to flush the current asynchronous BATCH
- using a BATCH in a dedicated thread will allow very fast background asynchronous process of ORM methods, sufficient for most use cases
- is a wrapper around BackgroundTimer.AsynchBatchStart()

**function AsynchBatchStop**

```
function AsynchBatchStop(Table: TSQLRecordClass): boolean;
```

*Finalize asynchronous ORM BATCH process, executed in a background thread*
- should have been preceded by a call to AsynchBatch(), or returns false
- Table=nil will release all existing batch instances
- is a wrapper around BackgroundTimer.AsynchBatchStop()
function AsynchBatchUpdate(Value: TSQLRecord; const CustomFields: TSQLFieldBits=[]; DoNotAutoComputeFields: boolean=false): integer;

*Update an ORM member in a BATCH to be written in a background thread*
- should have been preceded by a call to AsynchBatchStart(), or returns -1
- is a wrapper around BackgroundTimer.AsynchBatchUpdate()
- this method is thread-safe

function BatchSend(Batch: TSQLRestBatch): integer; overload;

*Execute a BATCH sequence prepared in a TSQLRestBatch instance*
- just a wrapper around the overloaded BatchSend() method without the Results: TIDDynArray parameter

function BatchSend(Batch: TSQLRestBatch; var Results: TIDDynArray): integer; overload; virtual;

*Execute a BATCH sequence prepared in a TSQLRestBatch instance*
- implements the "Unit Of Work" pattern, i.e. safe transactional process even on multi-thread environments
- send all pending Add/Update/Delete statements to the DB or remote server
- will return the URI Status value, i.e. 200/HTTP_SUCCESS OK on success
- a dynamic array of integers will be created in Results, containing all ROWDID created for each BatchAdd call, 200 (=HTTP_SUCCESS) for all successful BatchUpdate/BatchDelete, or 0 on error
- any error during server-side process MUST be checked against Results[] (the main URI Status is 200 if about communication success, and won't imply that all statements in the BATCH sequence were successful), or boRollbackOnError should be set in TSQLRestBatchOptions
- note that the caller shall still free the supplied Batch instance

function CacheOrNil: TSQLRestCache;

*Access the internal caching parameters for a given TSQLRecord*
- will return nil if no TSQLRestCache instance has been defined

class function ClassFrom(aDefinition: TSynConnectionDefinition): TSQLRestClass;

*Retrieve the registered class from the aDefinition.Kind string*

class function CreateFrom(aModel: TSQLModel; aDefinition: TSynConnectionDefinition): TSQLRest;

*Create a new TSQLRest instance from its Model and stored values*
- aDefinition.Kind will define the actual class which will be instantiated: currently TSQLRestServerFullMemory, TSQLRestServerDB, TSQLRestClientURINamedPipe, TSQLRestClientURIMessage, TSQLHttpClientWinSock, TSQLHttpClientWinInet, TSQLHttpClientWinHTTP, and TSQLHttpClientCurl classes are recognized by this method
- then other aDefinition fields will be used to refine the instance: please refer to each overridden DefinitionTo() method documentation
- use TSQLRestMongoDBCreate() and/or TSQLRestExternalDBCreate() instead to create a TSQLRest instance will all tables defined as external when aDefinition.Kind is 'MongoDB' or a TSQLDBConnectionProperties class
- will raise an exception if the supplied definition are not valid

class function CreateFromFile(aModel: TSQLModel; const aJSONFile: TFileName; aKey: cardinal=0): TSQLRest;

*Create a new TSQLRest instance from its Model and a JSON file*
- aDefinition.Kind will define the actual class which will be instantiated
- you can specify a custom Key, if the default is not safe enough for you
**Synopsis**

mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

---

**class function** CreateFromJSON(aModel: TSQLModel; const aJSONDefinition: RawUTF8; aKey: cardinal=0): TSQLRest;

*Create a new TSQLRest instance from its Model and JSON stored values*
- aDefinition.Kind will define the actual class which will be instantiated
- you can specify a custom Key, if the default is not safe enough for you

**class function** CreateTryFrom(aModel: TSQLModel; aDefinition: TSynConnectionDefinition; aServerHandleAuthentication: boolean): TSQLRest;

*Try to create a new TSQLRest instance from its Model and stored values*
- will return nil if the supplied definition are not valid
- if the newly created instance is a TSQLRestServer, will force the supplied aServerHandleAuthentication parameter to enable authentication

**function** DefinitionToJSON(Key: cardinal=0): RawUTF8;

*Save the properties into a JSON file*
- you can then use TSQLRest.CreateFromJSON() to re-instantiate it
- you can specify a custom Key, if the default is not enough for you

**function** Delete(Table: TSQLRecordClass; ID: TID): boolean; overload;

*Delete a member*
- implements REST DELETE collection
- return true on success
- call internally the EngineDelete() abstract method

**function** Delete(Table: TSQLRecordClass; const FormatSQLWhere: RawUTF8; const BoundsSQLWhere: array of const): boolean; overload;

*Delete a member with a WHERE clause*
- implements REST DELETE collection
- return true on success
- for better server speed, the WHERE clause should use bound parameters identified as '?' in the FormatSQLWhere statement, which is expected to follow the order of values supplied in BoundsSQLWhere open array - use DateToSQL/DateTimeToSQL for TDateTime, or directly any integer / double / currency / RawUTF8 values to be bound to the request as parameters
- is a simple wrapper around:
```
Delete(Table, FormatUTF8(FormatSQLWhere, [], BoundsSQLWhere))
```

**function** Delete(Table: TSQLRecordClass; const SQLWhere: RawUTF8): boolean; overload; virtual;

*Delete a member with a WHERE clause*
- implements REST DELETE collection
- return true on success
- this default method call OneFieldValues() to retrieve all matching IDs, then will delete each row using protected EngineDeleteWhere() virtual method

**function** Execute(const aSQL: RawUTF8): boolean; virtual;

*Execute directly a SQL statement, without any expected result*
- implements POST SQL on ModelRoot URI
- return true on success
- will call EngineExecute() abstract method to run the SQL statement
function ExecuteFmt(const SQLFormat: RawUTF8; const Args, Bounds: array of const): boolean; overload;

Execute directly a SQL statement with supplied parameters, with no result
- expect the same format as FormatUTF8() function, replacing all '%' chars with Args[] values,
  and all '?' chars with Bounds[] (inlining them with :(...) and auto-quotating strings)
- return true on success

function ExecuteFmt(const SQLFormat: RawUTF8; const Args: array of const): boolean; overload;

Execute directly a SQL statement with supplied parameters, with no result
- expect the same format as FormatUTF8() function, replacing all '%' chars with Args[] values
- return true on success

function ExecuteJson(const Tables: array of TSQLRecordClass; const SQL: RawUTF8; ForceAJAX: Boolean=false; ReturnedRowCount: PPtrInt=nil): RawJSON; virtual;

Execute directly a SQL statement, expecting a list of results
- you should not have to use this method, but the ORM versions instead
- return a result set as JSON on success, '' on failure
- will call EngineList() abstract method to retrieve its JSON content

function ExecuteList(const Tables: array of TSQLRecordClass; const SQL: RawUTF8): TSQLTableJSON; virtual;

Execute directly a SQL statement, expecting a list of results
- return a result table on success, nil on failure
- will call EngineList() abstract method to retrieve its JSON content

function FTSMatch(Table: TSQLRecordFTS3Class; const MatchClause: RawUTF8; var DocID: TIDDynArray; const PerFieldWeight: array of double; limit: integer=0; offset: integer=0): boolean; overload;

Dedicated method used to retrieve free-text matching DocIDs with enhanced ranking information
- this method works for TSQLRecordFTS3, TSQLRecordFTS4 and TSQLRecordFTS5
- this method will search in all FTS3 columns, and except some floating-point constants for
  weighting each column (there must be the same number of PerFieldWeight parameters as there
  are columns in the TSQLRecordFTS3 table)
- example of use: FTSMatch(TSQLDocuments,"linu*",IntResult,[1,0.5]) which will sort the
  results by the rank obtained with the 1st column/field being given twice the weighting of those
  in the 2nd (and last) column
- FTSMatch(TSQLDocuments,'linu*',IntResult,[1,0.5]) will perform a SQL query as such, which is
  the fastest way of ranking according to http://www.sqlite.org/fts3.html#appendix_a
  SELECT RowID FROM Documents WHERE Documents MATCH 'linu*' ORDER BY rank(matchinfo(Documents),1.0,0.5) DESC

function FTSMatch(Table: TSQLRecordFTS3Class; const WhereClause: RawUTF8; var DocID: TIDDynArray): boolean; overload;

Dedicated method used to retrieve free-text matching DocIDs
- this method works for TSQLRecordFTS3, TSQLRecordFTS4 and TSQLRecordFTS5
- this method expects the column/field names to be supplied in the MATCH statement clause
- example of use: FTSMatch(TSQLMessage,'Body MATCH :("linu**"):',IntResult) (using inlined
  parameters via :(...) is always a good idea)
function MainFieldID(Table: TSQLRecordClass; const Value: RawUTF8): TID;

  Return the ID of the record which main field match the specified value
  - search field is mainly the "Name" property, i.e. the one with "stored AS_UNIQUE" (i.e. "stored false") definition on most TSQLRecord
  - returns 0 if no matching record was found } 

function MainFieldIDs(Table: TSQLRecordClass; const Values: array of RawUTF8; out IDs: TIDDynArray): boolean;

  Return the IDs of the record which main field match the specified values
  - search field is mainly the "Name" property, i.e. the one with "stored AS_UNIQUE" (i.e. "stored false") definition on most TSQLRecord
  - if any of the Values[] is not existing, then no ID will appear in the IDs[] array - e.g. it will return [] if no matching record was found
  - returns TRUE if any matching ID was found (i.e. if length(IDs)>0) }

function MainFieldValue(Table: TSQLRecordClass; ID: TID; ReturnFirstIfNoUnique: boolean=false): RawUTF8;

  Retrieve the main field (mostly 'Name') value of the specified record
  - use GetMainFieldName() method to get the main field name
  - use OneFieldValue() method to get the field value
  - return '' if no such field or record exists
  - if ReturnFirstIfNoUnique is TRUE and no unique property is found, the first RawUTF8 property is returned anyway

function MemberExists(Table: TSQLRecordClass; ID: TID): boolean; virtual;

  Check if a given ID do exist for a given table

function MultiFieldValue(Table: TSQLRecordClass; const FieldName: array of RawUTF8; var FieldValue: array of RawUTF8; const WhereClause: RawUTF8): boolean; overload;

  Get the UTF-8 encoded value of some fields with a Where Clause
  - example of use: MultiFieldValue(TSQLRecord,['Name',Name,'ID=:(23):']) (using inlined parameters via :(...): is always a good idea)
  - FieldValue[] will have the same length as FieldName[]
  - return true on success, false on SQL error or no result
  - call internally ExecuteList() to get the list

function MultiFieldValue(Table: TSQLRecordClass; const FieldName: array of RawUTF8; var FieldValue: array of RawUTF8; WhereID: TID): boolean; overload;

  Get the UTF-8 encoded value of some fields from its ID
  - example of use: MultiFieldValue(TSQLRecord,['Name',Name,23)
  - FieldValue[] will have the same length as FieldName[]
  - return true on success, false on SQL error or no result
  - call internally ExecuteList() to get the list
function MultiFieldValues(Table: TSQLRecordClass; const FieldNames: RawUTF8; const WhereClause: RawUTF8=''): TSQLTableJSON; overload; virtual;

Execute directly a SQL statement, expecting a list of results
- return a result table on success, nil on failure
- FieldNames can be the CSV list of field names to be retrieved
  - if FieldNames is "," will get all simple fields, excluding BLOBs
  - if FieldNames is "*", will get all fields, including ID and BLOBs
- call internal ExecuteList() to get the list
- using inlined parameters via :(...): in WhereClause is always a good idea

function MultiFieldValues(Table: TSQLRecordClass; const FieldNames: RawUTF8; const WhereClauseFormat: RawUTF8; const BoundsSQLWhere: array of const): TSQLTableJSON; overload;

Execute directly a SQL statement, expecting a list of results
- return a result table on success, nil on failure
- FieldNames can be the CSV list of field names to be retrieved
  - if FieldNames is "," will get all simple fields, excluding BLOBs
  - if FieldNames is "*", will get all fields, including ID and BLOBs
- this overloaded function will call FormatUTF8 to create the Where Clause from supplied parameters, binding all '?' chars with Args[] values
  - example of use:
    aList := aClient.MultiFieldValues(TSQLRecord, 'Name,FirstName', 'Salary>=?', [aMinSalary]);
  - call overloaded MultiFieldValues() / ExecuteList() to get the list
  - note that this method prototype changed with revision 1.17 of the framework: array of const used to be Args and '%' in the WhereClauseFormat statement, whereas it now expects bound parameters as '?'

function MultiFieldValues(Table: TSQLRecordClass; const FieldNames: RawUTF8; const WhereClauseFormat: RawUTF8; const Args, Bounds: array of const): TSQLTableJSON; overload;

Execute directly a SQL statement, expecting a list of results
- return a result table on success, nil on failure
- FieldNames can be the CSV list of field names to be retrieved
  - if FieldNames is "," will get all simple fields, excluding BLOBs
  - if FieldNames is "*", will get all fields, including ID and BLOBs
- in this version, the WHERE clause can be created with the same format as FormatUTF8() function, replacing all '%' chars with Args[], and all '?' chars with Bounds[] (inlining them with :(...): and auto-quoting strings)
  - example of use:
    Table := MultiFieldValues(TSQLRecord, 'Name', '%=?', ['ID'], [aID]);
  - call overloaded MultiFieldValues() / ExecuteList() to get the list
function MultiRedirect(const aGUID: TGUID; out aCallbackInterface; 
aCallBackUnRegisterNeeded: boolean=true): IMultiCallbackRedirect; overload;

- Define redirection of interface methods calls in one or several instances
- this class allows to implements any interface via a fake class, which will redirect all methods 
calls to one or several other interfaces
- returned aCallbackInterface will redirect all its methods (identified by aGUID) into an internal 
list handled by IMultiCallbackRedirect.Redirect
- typical use thefore:
fSharedCallback: IMyService;
fSharedCallbacks: IMultiCallbackRedirect;
... 
if fSharedCallbacks=nil then begin
  fSharedCalls := aRest.MultiRedirect(IMyService,fSharedCallback);
aServices.SubscribeForEvents(fSharedCallback);
end;
fSharedCalls.Redirect(TMyCallback.Create,[]);
// now each time fSharedCallback receive one event, all callbacks 
// previously registered via Redirect() will receive it
...
fSharedCalls := nil; // will stop redirection 
// and unregister callbacks, if needed

function NewBackgroundThreadMethod(const Format: RawUTF8; const Args: array of 
  const): TSynBackgroundThreadMethod;

- Allows to safely execute a processing method in a background thread
- returns a TSynBackgroundThreadMethod instance, ready to execute any background task via 
its RunAndWait() method
- will properly call BeginCurrentThread/EndCurrentThread methods
- you should supply some runtime information to name the thread, for proper debugging

function NewBackgroundThreadProcess(aOnProcess: TOnSynBackgroundThreadProcess;
aOnProcessMS: cardinal; const Format: RawUTF8; const Args: array of const; aStats: 
  TSynMonitorClass=nil): TSynBackgroundThreadProcess;

- Allows to safely execute a process at a given pace
- returns a TSynBackgroundThreadProcess instance, ready to execute the supplied aOnProcess 
event in a loop, as aOnProcessMS periodic task
- will properly call BeginCurrentThread/EndCurrentThread methods
- you should supply some runtime information to name the thread, for proper debugging

function NewParallelProcess(ThreadCount: integer; const Format: RawUTF8; const 
  Args: array of const): TSynParallelProcess;

- Allows to safely execute a process in parallel
- returns a TSynParallelProcess instance, ready to execute any task in parallel in a thread-pool 
given by ThreadCount
- will properly call BeginCurrentThread/EndCurrentThread methods
- you should supply some runtime information to name the thread, for proper debugging

function OneFieldValue(Table: TSQLRecordClass; const FieldName: RawUTF8; WhereID: 
  TID): RawUTF8; overload;

- Get the UTF-8 encoded value of an unique field from its ID
- example of use: OneFieldValue(TSQLRecord,'Name',23)
- call internally ExecuteList() to get the value
function OneFieldValue(Table: TSQLRecordClass; const FieldName: RawUTF8; const WhereClauseFmt: RawUTF8; const Args, Bounds: array of const; out Data: Int64): boolean; overload;

- Get one integer value of an unique field with a Where Clause
- this overloaded function will return the field value as integer

function OneFieldValue(Table: TSQLRecordClass; const FieldName, WhereClause: RawUTF8): RawUTF8; overload;

- Get the UTF-8 encoded value of an unique field with a Where Clause
- example of use - including inlined parameters via :(...):
  aClient.OneToOneFieldValue(TSQLRecord,'Name','ID=:(23):')

  you should better call the corresponding overloaded method as such:
  aClient.OneToOneFieldValue(TSQLRecord,'Name','ID=?',[aID])

  which is the same as calling:
  aClient.OneToOneFieldValue(TSQLRecord,'Name',FormatUTF8('ID=?',[],[23]))

  - call internaly ExecuteList() to get the value

function OneFieldValue(Table: TSQLRecordClass; const FieldName: RawUTF8; const FormatSQLWhere: RawUTF8; const BoundsSQLWhere: array of const): RawUTF8; overload;

- Get the UTF-8 encoded value of an unique field with a Where Clause
- this overloaded function will call FormatUTF8 to create the Where Clause from supplied parameters, binding all '?' chars with Args[] values
- example of use:
  aClient.OneToOneFieldValue(TSQLRecord,'Name','ID=?',[aID])

  - call internaly ExecuteList() to get the value

  - note that this method prototype changed with revision 1.17 of the framework: array of const used to be Args and '%' in the FormatSQLWhere statement, whereas it now expects bound parameters as '?'

function OneFieldValue(Table: TSQLRecordClass; const FieldName: RawUTF8; const WhereClauseFmt: RawUTF8; const Args, Bounds: array of const): RawUTF8; overload;

- Get the UTF-8 encoded value of an unique field with a Where Clause
- this overloaded function will call FormatUTF8 to create the Where Clause from supplied parameters, replacing all '%' chars with Args[], and all '?' chars with Bounds[] (inlining them with :(...): and auto-quoting strings)
- example of use:
  OneFieldValue(TSQLRecord,'Name','%=?',['ID'],[aID])

  - call internaly ExecuteList() to get the value

function OneFieldValueInt64(Table: TSQLRecordClass; const FieldName, WhereClause: RawUTF8; Default: Int64=0): Int64;

- Get the Int64 value of an unique field with a Where Clause
- call internaly ExecuteList() to get the value

function OneFieldValues(Table: TSQLRecordClass; const FieldName: RawUTF8; const WhereClauseFmt: RawUTF8=''; const Separator: RawUTF8=' ','): RawUTF8; overload;

- Get the CSV-encoded UTF-8 encoded values of an unique field with a Where Clause
- example of use: OneFieldValues(TSQLRecord,'FirstName','Name=('Smith'),Data) (using inlined parameters via :(...): is always a good idea)
- leave WhereClause void to get all records
- call internaly ExecuteList() to get the list
- using inlined parameters via :(...): in WhereClause is always a good idea
function OneFieldValues(Table: TSQLRecordClass; const FieldName, WhereClause: RawUTF8; Strings: TStrings; IDToIndex: PID=nil): Boolean; overload;

Get the string-encoded values of an unique field into some TStrings
- Items[] will be filled with string-encoded values of the given field
- Objects[] will be filled with pointer(ID)
- call internally ExecuteList() to get the list
- returns TRUE on success, FALSE if no data was retrieved
- if IDToIndex is set, its value will be replaced with the index in Strings.Objects[] where ID=IDToIndex^
- using inlined parameters via :(...):: in WhereClause is always a good idea

function OneFieldValues(Table: TSQLRecordClass; const FieldName: RawUTF8; const WhereClause: RawUTF8; var Data: TInt64DynArray; SQL: PRawUTF8=nil): boolean; overload;

Get the integer value of an unique field with a Where Clause
- example of use: OneFieldValue(TSQLRecordPeopel,'ID','Name=:("Smith")::,Data) (using inlined parameters via :(...):: is always a good idea)
- leave WhereClause void to get all records
- call internally ExecuteList() to get the list

function OneFieldValues(Table: TSQLRecordClass; const FieldName: RawUTF8; const WhereClause: RawUTF8; out Data: TRawUTF8DynArray): boolean; overload;

Get the UTF-8 encoded values of an unique field with a Where Clause
- example of use: OneFieldValue(TSQLRecord,'FirstName','Name=:("Smith")::,Data) (using inlined parameters via :(...):: is always a good idea)
- leave WhereClause void to get all records
- call internally ExecuteList() to get the list
- returns TRUE on success, FALSE if no data was retrieved

class function QueryIsTrue(aTable: TSQLRecordClass; aID: TID; FieldType: TSQLFieldType; Value: PUTF8Char; Operator: integer; Reference: PUTF8Char): boolean;

Evaluate a basic operation for implementing User Interface Query action
- expect both Value and Reference to be UTF-8 encoded (as in TSQLTable or TSQLTableToGrid)
- aID parameter is ignored in this function implementation (expect only this parameter to be not equal to 0)
- is TSQLQueryEvent prototype compatible
- for qoContains and qoBeginWith, the Reference is expected to be already uppercase
- for qoSoundsLike* operators, Reference is not a PUTF8Char, but a typecase of a prepared TSynSoundEx object instance (i.e. pointer(@SoundEx))

function RecordCanBeUpdated(Table: TSQLRecordClass; ID: TID; Action: TSQLEvent; ErrorMsg: PRawUTF8 = nil): boolean; virtual;

Override this method to guess if this record can be updated or deleted
- this default implementation returns always true
- e.g. you can add digital signature to a record to disallow record editing
- the ErrorMsg can be set to a variable, which will contain an explicit error message

mORMot.pas unit - Rev. 1.18   Page 2119 of 2552
function Retrieve(Reference: TRecordReference; ForUpdate: boolean=false): TSQLRecord; overload; virtual;

*Get a member from its TRecordReference property content*
- instead of the other Retrieve() methods, this implementation Create an instance, with the appropriated class stored in Reference
- returns nil on any error (invalid Reference e.g.)
- if ForUpdate is true, the REST method is LOCK and not GET: it tries to lock the corresponding record, then retrieve its content; caller has to call UnLock() method after Value usage, to release the record
- the TSQLRawBlob (BLOB) fields are not retrieved by this method, to preserve bandwidth: use the RetrieveBlob() methods for handling BLOB fields, or set either the TSQLRestClientURI.ForceBlobTransfert or TSQLRestClientURI.ForceBlobTransfertTable[] properties
- the TSQLRecordMany fields are not retrieved either: they are separate instances created by TSQLRecordMany.Create, with dedicated methods to access to the separated pivot table

function Retrieve(aPublishedRecord, aValue: TSQLRecord): boolean; overload;

*Get a member from a published property TSQLRecord*
- those properties are not class instances, but TObject(aRecordID)
- is just a wrapper around Retrieve(aPublishedRecord.ID,aValue)
- return true on success

function Retrieve(aID: TID; Value: TSQLRecord; ForUpdate: boolean=false): boolean; overload; virtual;

*Get a member from its ID*
- return true on success
- Execute 'SELECT * FROM TableName WHERE ID=:(aID): LIMIT 1' SQL Statement
- if ForUpdate is true, the REST method is LOCK and not GET: it tries to lock the corresponding record, then retrieve its content; caller has to call UnLock() method after Value usage, to release the record
- this method will call EngineRetrieve() abstract method
- the TSQLRawBlob (BLOB) fields are not retrieved by this method, to preserve bandwidth: use the RetrieveBlob() methods for handling BLOB fields, or set either the TSQLRestClientURI.ForceBlobTransfert or TSQLRestClientURI.ForceBlobTransfertTable[] properties
- the TSQLRecordMany fields are not retrieved either: they are separate instances created by TSQLRecordMany.Create, with dedicated methods to access to the separated pivot table

function Retrieve(const SQLWhere: RawUTF8; Value: TSQLRecord; const aCustomFieldsCSV: RawUTF8=''): boolean; overload; virtual;

*Here are REST basic direct calls (works with Server or Client) get a member from a SQL statement*
- implements REST GET collection
- return true on success
- Execute 'SELECT * FROM TableName WHERE SQLWhere LIMIT 1' SQL Statement (using inlined parameters via :(...) in SQLWhere is always a good idea)
- since no record is specified, locking is pointless here
- default implementation call ExecuteList(), and fill Value from a temporary TSQLTable
- if aCustomFieldsCSV is ", will get all simple fields, excluding BLOBs and TSQLRecordMany fields (use RetrieveBlob method or set TSQLRestClientURI.ForceBlobTransfert)
- if aCustomFieldsCSV is "*, will get ALL fields, including ID and BLOBs
- if this default set of simple fields does not fit your need, you could specify your own set
function Retrieve(const WhereClauseFmt: RawUTF8; const Args,Bounds: array of const; Value: TSQLRecord; const aCustomFieldsCSV: RawUTF8=''): boolean; overload;

Get a member from a SQL statement
- implements REST GET collection
- return true on success
- same as Retrieve(const SQLWhere: RawUTF8; Value: TSQLRecord) method, but this overloaded function will call FormatUTF8 to create the Where Clause from supplied parameters, replacing all '%' chars with Args[], and all '?' chars with Bounds[] (inlining them with :(...): and auto-quoting strings)
- if aCustomFieldsCSV is '', will get all simple fields, excluding BLOBs
- if aCustomFieldsCSV is '*', will get ALL fields, including ID and BLOBs

function RetrieveBlob(Table: TSQLRecordClass; aID: TID; const BlobFieldName: RawUTF8; out BlobStream: THeapMemoryStream): boolean; overload;

Get a blob field content from its record ID and supplied blob field name
- implements REST GET collection with a supplied member ID and field name
- return true on success
- this method will create a TStream instance (which must be freed by the caller after use) and fill it with the blob data

function RetrieveBlob(Table: TSQLRecordClass; aID: TID; const BlobFieldName: RawUTF8; const BlobData: TSQLRawBlob): boolean; overload;

virtual;

Get a blob field content from its record ID and supplied blob field name
- implements REST GET collection with a supplied member ID and a blob field name
- return true on success
- this method is defined as abstract, i.e. there is no default implementation: it must be implemented 100% RestFul with a GET ModelRoot/TableName/TableID/BlobFieldName request for example
- this method retrieve the blob data as a TSQLRawBlob string using EngineRetrieveBlob()

function RetrieveBlobFields(Value: TSQLRecord): boolean; virtual;

Get all BLOB fields of the supplied value from the remote server
- call several REST GET collection (one for each BLOB) for the member
- call interanly e.g. by TSQLRestClient.Retrieve method when ForceBlobTransfert / ForceBlobTransfertTable[] is set

function RetrieveDocVariant(Table: TSQLRecordClass; const FormatSQLWhere: RawUTF8; const BoundsSQLWhere: array of const; const CustomFieldsCSV: RawUTF8): variant;

Get one member from a SQL statement as a TDocVariant
- implements REST GET collection
- the data will be converted to a TDocVariant variant following the TSQLRecord layout, so complex types like dynamic array will be returned as a true array of values
function RetrieveDocVariantArray(Table: TSQLRecordClass; const ObjectName, CustomFieldsCSV: RawUTF8; FirstRecordID: PID=nil; LastRecordID: PID=nil): variant; overload;

Get a list of all members from a SQL statement as a TDocVariant
- implements REST GET collection
- if ObjectName='', it will return a TDocVariant of dvArray kind
- if ObjectName is set, it will return a TDocVariant of dvObject kind, with one property containing the array of values: this returned variant can be pasted e.g. directly as parameter to TSynMustache.Render()
- aCustomFieldsCSV can be the CSV list of field names to be retrieved
- if aCustomFieldsCSV is '', will get all simple fields, excluding BLOBs
- if aCustomFieldsCSV is '*', will get ALL fields, including ID and BLOBs
- the data will be converted to variants and TDocVariant following the TSQLRecord layout, so complex types like dynamic array will be returned as a true array of values (in contrast to the RetrieveListJSON method)
- warning: under FPC, we observed that assigning the result of this method to a local variable may circumvent a memory leak FPC bug

function RetrieveDocVariantArray(Table: TSQLRecordClass; const ObjectName: RawUTF8; const FormatSQLWhere: RawUTF8; const BoundsSQLWhere: array of const; const CustomFieldsCSV: RawUTF8; FirstRecordID: PID=nil; LastRecordID: PID=nil): variant; overload;

Get a list of members from a SQL statement as a TDocVariant
- implements REST GET collection over a specified WHERE clause
- if ObjectName='', it will return a TDocVariant of dvArray kind
- if ObjectName is set, it will return a TDocVariant of dvObject kind, with one property containing the array of values: this returned variant can be pasted e.g. directly as parameter to TSynMustache.Render()
- for better server speed, the WHERE clause should use bound parameters identified as '?' in the FormatSQLWhere statement, which is expected to follow the order of values supplied in BoundsSQLWhere open array - use DateToSQL()/DateTimeToSQL() for TDateTime, or directly any integer, double, currency, RawUTF8 values to be bound to the request as parameters
- aCustomFieldsCSV can be the CSV list of field names to be retrieved
- if aCustomFieldsCSV is '', will get all simple fields, excluding BLOBs
- if aCustomFieldsCSV is '*', will get ALL fields, including ID and BLOBs
- the data will be converted to variants and TDocVariant following the TSQLRecord layout, so complex types like dynamic array will be returned as a true array of values (in contrast to the RetrieveListJSON method)
- warning: under FPC, we observed that assigning the result of this method to a local variable may circumvent a memory leak FPC bug
function RetrieveList(Table: TSQLRecordClass; const FormatSQLWhere: RawUTF8; const BoundsSQLWhere: array of const; const aCustomFieldsCSV: RawUTF8=''): TObjectList; overload;

Get a list of members from a SQL statement as TObjectList
- implements REST GET collection
- for better server speed, the WHERE clause should use bound parameters identified as '?' in the FormatSQLWhere statement, which is expected to follow the order of values supplied in BoundsSQLWhere open array - use DateToSQL()/DateTimeToSQL() for TDateTime, or directly any integer, double, currency, RawUTF8 values to be bound to the request as parameters
- aCustomFieldsCSV can be the CSV list of field names to be retrieved
  - if aCustomFieldsCSV is '', will get all simple fields, excluding BLOBs
  - if aCustomFieldsCSV is '*', will get ALL fields, including ID and BLOBs
- return a TObjectList on success (possibly with Count=0) - caller is responsible of freeing the instance
  - this TObjectList will contain a list of all matching records
- return nil on error

function RetrieveListJSON(Table: TSQLRecordClass; const FormatSQLWhere: RawUTF8; const BoundsSQLWhere: array of const; const aCustomFieldsCSV: RawUTF8=''; aForceAJAX: boolean=false): RawJSON; overload;

Get a list of members from a SQL statement as RawJSON
- implements REST GET collection
- for better server speed, the WHERE clause should use bound parameters identified as '?' in the FormatSQLWhere statement, which is expected to follow the order of values supplied in BoundsSQLWhere open array - use DateToSQL()/DateTimeToSQL() for TDateTime, or directly any integer, double, currency, RawUTF8 values to be bound to the request as parameters
- aCustomFieldsCSV can be the CSV list of field names to be retrieved
  - if aCustomFieldsCSV is '', will get all simple fields, excluding BLOBs
  - if aCustomFieldsCSV is '*', will get ALL fields, including ID and BLOBs
- returns the raw JSON array content with all items on success, with our expanded / not expanded JSON format - so can be used with SOA methods and RawJSON results, for direct process from the client side
- returns '' on error
- the data is directly retrieved from raw JSON as returned by the database without any conversion, so this method will be the fastest, but complex types like dynamic array will be returned as Base64-encoded blob value - if you need proper JSON access to those, see RetrieveDocVariantArray()
function RetrieveListJSON(Table: TSQLRecordClass; const SQLWhere: RawUTF8; const aCustomFieldsCSV: RawUTF8=''; aForceAJAX: boolean=false): RawJSON; overload;

Get a list of members from a SQL statement as RawJSON
- implements REST GET collection
- this overloaded version expect the SQLWhere clause to be already prepared with inline parameters using a previous FormatUTF8() call
- aCustomFieldsCSV can be the CSV list of field names to be retrieved
- if aCustomFieldsCSV is '', will get all simple fields, excluding BLOBs
- if aCustomFieldsCSV is '*', will get ALL fields, including ID and BLOBs
- returns the raw JSON array content with all items on success, with our expanded / not expanded JSON format - so can be used with SOA methods and RawJSON results, for direct process from the client side
- returns '' on error
- the data is directly retrieved from raw JSON as returned by the database without any conversion, so this method will be the fastest, but complex types like dynamic array will be returned as Base64-encoded blob value - if you need proper JSON access to those, see RetrieveDocVariantArray()

function RetrieveListObjArray(var ObjArray: T*ObjArray; Table: TSQLRecordClass; const FormatSQLWhere: RawUTF8; const BoundsSQLWhere: array of const; const aCustomFieldsCSV: RawUTF8=''): boolean;

Get a list of members from a SQL statement as T*ObjArray
- implements REST GET collection
- for better server speed, the WHERE clause should use bound parameters identified as '?' in the FormatSQLWhere statement, which is expected to follow the order of values supplied in BoundsSQLWhere open array - use DateToSQL()/DateTimeToSQL() for TDateTime, or directly any integer, double, currency, RawUTF8 values to be bound to the request as parameters
- aCustomFieldsCSV can be the CSV list of field names to be retrieved
- if aCustomFieldsCSV is '', will get all simple fields, excluding BLOBs
- if aCustomFieldsCSV is '*', will get ALL fields, including ID and BLOBs
- set the T*ObjArray variable with all items on success - so that it can be used with SOA methods
- it is up to the caller to ensure that ObjClear(ObjArray) is called when the T*ObjArray list is not needed any more
- returns true on success, false on error

function RetrieveOneFieldDocVariantArray(Table: TSQLRecordClass; const FieldName, FormatSQLWhere: RawUTF8; const BoundsSQLWhere: array of const): variant;

Get all values of a SQL stateme}nt on a single column as a TDocVariant array
- implements REST GET collection on a single field
- for better server speed, the WHERE clause should use bound parameters identified as '?' in the FormatSQLWhere statement, which is expected to follow the order of values supplied in BoundsSQLWhere open array - use DateToSQL()/DateTimeToSQL() for TDateTime, or directly any integer, double, currency, RawUTF8 values to be bound to the request as parameters
- the data will be converted to variants and TDocVariant following the TSQLRecord layout, so complex types like dynamic array will be returned as a true array of values (in contrast to the RetrieveListJSON method)
function RTreeMatch(DataTable: TSQLRecordClass; const DataTableBlobFieldName: RawUTF8; RTreeTable: TSQLRecordRTreeClass; const DataTableBlobField: RawByteString; var DataID: TIDDynArray): boolean;

Dedicated method used to retrieve matching IDs using a fast R-Tree index
- a TSQLRecordRTree is associated to a TSQLRecord with a specified BLOB field, and will call TSQLRecordRTree BlobToCoord and ContainedIn virtual class methods to execute an optimized SQL query
  - as alternative, with SQLite3 >= 3.24.0, you may use Auxiliary Columns
  - will return all matching DataTable IDs in DataID[]
  - will generate e.g. the following statement
    SELECT MapData.ID From MapData, MapBox WHERE MapData.ID=MapBox.ID
    AND minX>=:-81.0: AND maxX<=:-79.6: AND minY>=:(35.0): AND :(maxY<=36.2):
    AND MapBox_in(MapData.BlobField,('--uFFF0base64encoded-81,-79.6,35,36.2'));
  
when the following Delphi code is executed:
  aClient.RTreeMatch(TSQLRecordMapData,'BlobField',TSQLRecordMapBox, aMapData.BlobField,ResultID);

function ServiceContainer: TServiceContainer; virtual; abstract;

Access or initialize the internal IoC resolver, used for interface-based remote services, and more generally any Services.Resolve() call
- create and initialize the internal TServiceContainer if no service interface has been registered yet // - may be used to inject some dependencies, which are not interface-based remote services, but internal IoC, without the ServiceRegister() or ServiceDefine() methods - e.g. aRest.ServiceContainer.InjectResolver([[TInfraRepoUserFactory.Create(aRest)],true]);
- overridden methods will return TServiceContainerClient or TServiceContainerServer instances, on TSQLRestClient or TSQLRestServer

function SystemUseTrack(periodSec: integer=10): TSystemUse;

Will gather CPU and RAM information in a background thread
- you can specify the update frequency, in seconds
- access to the information via the returned instance, which maps the TSystemUse.Current class function
- do nothing if global TSystemUse.Current was already assigned

function TableHasRows(Table: TSQLRecordClass): boolean; virtual;

Check if there is some data rows in a specified table
- calls internally a "SELECT RowID FROM TableName LIMIT 1" SQL statement, which is much faster than testing if "SELECT count(*)" equals 0 - see @http://stackoverflow.com/questions/8988915

function TableMaxID(Table: TSQLRecordClass): TID; virtual;

Search for the last inserted ID in a specified table
- returns -1 on error
- will execute by default "SELECT max(rowid) FROM TableName"

function TableRowCount(Table: TSQLRecordClass): Int64; virtual;

Get the row count of a specified table
- returns -1 on error
- returns the row count of the table on success
- calls internally the "SELECT Count(*) FROM TableName;" SQL statement
**function** TimerDisable(aOnProcess: TOnSynBackgroundTimerProcess): boolean;

*Undefine a task running on a periodic number of seconds*
- should have been registered by a previous call to TimerEnable() method
- returns true on success, false if the supplied task was not registered

**function** TimerEnable(aOnProcess: TOnSynBackgroundTimerProcess; aOnProcessSecs: cardinal): TSynBackgroundTimer;

*Define a task running on a periodic number of seconds in a background thread*
- could be used to run background maintenance or monitoring tasks on this TSQLEst instance, at a low pace (typically every few minutes)
- will instantiate and run a shared TSynBackgroundTimer instance for this TSQLEst, so all tasks will share the very same thread
- you can run BackgroundTimer.Enqueue or ExecuteNow methods to implement a FIFO queue, or force immediate execution of the process
- will call BeginCurrentThread/EndCurrentThread as expected e.g. by logs

**function** TransactionActiveSession: cardinal;

*Check current transaction status*
- returns the session ID if a transaction is active
- returns 0 if no transaction is active
### function TransactionBegin(aTable: TSQLRecordClass; SessionID: cardinal): boolean;

**virtual**;

*Begin a transaction*
- implements REST BEGIN collection
- may be used to speed up CRUD statements like Add/Update/Delete
- in the current implementation, nested transactions are not allowed
- must be ended with Commit on success
- must be aborted with Rollback if any SQL statement failed
- default implementation just handle the protected fTransactionActiveSession flag
- return true if no transaction is active, false otherwise
- in aClient-Server environment with multiple Clients connected at the same time, you should better use BATCH process, specifying a positive AutomaticTransactionPerRow parameter to BatchStart()
- in a multi-threaded or Client-Server with multiple concurrent Client connections, you may check the returned value, as such:

```pascal
if Client.TransactionBegin(TSQLRecordPeopleObject) then
  try
    //.... modify the database content, raise exceptions on error
    Client.Commit;
  except
  Client.Rollback; // in case of error
end;
```

or use the TransactionBeginRetry() method
- the supplied SessionID will allow multi-user transaction safety on the Server-Side: all database modification from another session will wait for the global transaction to be finished; on Client-side, the SessionID is just ignored (TSQLRestClient will override this method with a default SessionID=CONST_AUTHENTICATION_NOT_USED=1 parameter)
- if you have an external database engine which expect transactions to take place in the same thread, ensure TSQLRestServer force execution of this method when accessed from RESTful clients in the same thread, e.g.:

```pascal
AcquireExecutionMode[execORMWrite] := amBackgroundThread;
AcquireWriteMode := amBackgroundThread; // same as previous
```

### function Unlock(Rec: TSQLRecord): boolean; overload;

*Unlock the corresponding record*
- record should have been locked previously e.g. with Retrieve() and forupdate=true, i.e. retrieved not via GET with LOCK REST-like verb
- use our custom UNLOCK REST-like method
- calls internally Unlock() above
- returns true on success

### function Unlock(Table: TSQLRecordClass; aID: TID): boolean; overload; virtual;

*abstract*;

*Unlock the corresponding record*
- record should have been locked previously e.g. with Retrieve() and forupdate=true, i.e. retrieved not via GET with LOCK REST-like verb
- use our custom UNLOCK REST-like verb
- returns true on success
function Update(aTable: TSQLRecordClass; aID: TID; const aSimpleFields: array of const): boolean; overload;

  *Update a member from a supplied list of simple field values*
  - implements REST PUT collection
  - the aSimpleFields parameters MUST follow explicetly both count and order of published properties of the supplied aTable class, excepting the TSQLRawBlob and TSQLRecordMany kind (i.e. only so called "simple fields")
  - return true on success
  - call internaly the Update() / EngineUpdate() virtual methods

function Update(Value: TSQLRecord; const CustomCSVFields: RawUTF8; DoNotAutoComputeFields: boolean=false): boolean; overload;

  *Update a member from Value simple fields content*
  - implements REST PUT collection
  - return true on success
  - is an overloaded method to Update(Value,FieldBitsFromCSV())

function Update(Value: TSQLRecord; const CustomFields: TSQLFieldBits=[]; DoNotAutoComputeFields: boolean=false): boolean; overload; virtual;

  *Update a member from Value simple fields content*
  - implements REST PUT collection
  - return true on success
  - the TSQLRawBlob(BLOB) fields values are not updated by this method, to preserve bandwidth:
    * use the UpdateBlob() methods for handling BLOB fields
  - the TSQLRecordMany fields are not set either: they are separate instances created by TSQLRecordMany.Create, with dedicated methods to access to the separated pivot table
  - if CustomFields is left void, the simple fields will be used, or the fields retrieved via a previous FillPrepare() call; otherwise, you can specify your own set of fields to be transmitted (including BLOBs, even if they will be Base64-encoded within the JSON content) - CustomFields could be computed by TSQLRecordProperties.FieldBitsFromCSV() or TSQLRecordProperties.FieldBitsFromRawUTF8()
  - this method will always compute and send any TModTime fields
  - this method will call EngineUpdate() to perform the request

function UpdateBlob(Table: TSQLRecordClass; aID: TID; const BlobFieldName: RawUTF8; const BlobData: TSQLRawBlob): boolean; overload; virtual;

  *Update a blob field from its record ID and supplied blob field name*
  - implements REST PUT collection with a supplied member ID and field name
  - return true on success
  - call internaly the EngineUpdateBlob() abstract method
  - this method expect the Blob data to be supplied as TSQLRawBlob, using EngineUpdateBlob()

function UpdateBlob(Table: TSQLRecordClass; aID: TID; const BlobFieldName: RawUTF8; BlobData: TStream): boolean; overload;

  *Update a blob field from its record ID and blob field name*
  - implements REST PUT collection with a supplied member ID and field name
  - return true on success
  - call internaly the EngineUpdateBlob() abstract method
  - this method expect the Blob data to be supplied as a TStream: it will send the whole stream content (from its beginning position upto its current size) to the database engine
function UpdateBlob(Table: TSQLRecordClass; aID: TID; const BlobFieldName: RawUTF8; BlobData: pointer; BlobSize: integer): boolean; overload;

Update a blob field from its record ID and blob field name
- implements REST PUT collection with a supplied member ID and field name
- return true on success
- call internally the EngineUpdateBlob() abstract method
- this method expect the Blob data to be supplied as direct memory pointer and size

function UpdateBlobFields(Value: TSQLRecord): boolean; virtual;

Update all BLOB fields of the supplied Value
- call several REST PUT collection (one for each BLOB) for the member
- uses the UpdateBlob() method to send the BLOB properties content to the Server
- called internally by Add and Update methods when ForceBlobTransfert / ForceBlobTransfertTable[] is set
- you can use this method by hand, to avoid several calls to UpdateBlob()
- returns TRUE on success (or if there is no BLOB field)
- returns FALSE on error (e.g. if Value is invalid or with db/transmission)

function UpdateField(Table: TSQLRecordClass; const WhereFieldName: RawUTF8; const WhereFieldValue: variant; const FieldName: RawUTF8; const FieldValue: variant): boolean; overload; virtual;

Update one field in one or several members, depending on a WHERE clause, with both update and where values specified as variant
- implements REST PUT collection with one field value on a one where value
- any value can be set in FieldValue, but for BLOBs, you should better use UpdateBlob()
- for security reasons, void WHERE clause will be rejected
- return true on success
- call internally the EngineUpdateField() abstract method
- note that this method won’t update the TModTime properties, nor the internal table Cache: you should rather use a classic Retrieve()/FillPrepare() followed by an Update() of the whole record

function UpdateField(Table: TSQLRecordClass; ID: TID; const FieldName: RawUTF8; const FieldValue: array of const): boolean; overload; virtual;

Update one field/column value a given member
- implements REST PUT collection with one field value
- only one single field shall be specified in FieldValue, but could be of any kind of value - for BLOBs, you should better use UpdateBlob()
- return true on success
- call internally the EngineUpdateField() abstract method
- note that this method won’t update the TModTime properties: you should rather use a classic Retrieve()/FillPrepare() followed by Update(); but it will notify the internal Cache
function UpdateField(Table: TSQLRecordClass; const WhereFieldName: RawUTF8; const WhereFieldValue: array of const; const FieldName: RawUTF8; const FieldValue: array of const): boolean; overload; virtual;

_Update one field in one or several members, depending on a WHERE clause_
- implements REST PUT collection with one field value on a one where value
- only one single field shall be specified in FieldValue, but could be of any kind of value - for BLOBs, you should better use UpdateBlob()
- only one single field shall be specified in WhereFieldValue, but could be of any kind of value - for security reasons, void WHERE clause will be rejected
- return true on success
- call internally the EngineUpdateField() abstract method
- note that this method won't update the TModTime properties: you should rather use a classic Retrieve()/FillPrepare() followed by Update(); but it will notify the internal Cache

function UpdateField(Table: TSQLRecordClass; ID: TID; const FieldName: RawUTF8; const FieldValue: variant): boolean; overload; virtual;

_Update one field in a given member with a value specified as variant_
- implements REST PUT collection with one field value
- any value can be set in FieldValue, but for BLOBs, you should better use UpdateBlob()
- return true on success
- call internally the EngineUpdateField() abstract method
- note that this method won't update the TModTime properties: you should rather use a classic Retrieve()/FillPrepare() followed by Update(); but it will notify the internal Cache

function UpdateField(Table: TSQLRecordClass; const IDs: array of Int64; const FieldName: RawUTF8; const FieldValue: variant): boolean; overload; virtual;

_Update one field in one or several members, depending on a set of IDs_
- return true on success
- note that this method won't update the TModTime properties: you should rather use a classic Retrieve()/FillPrepare() followed by Update(), but it will be much slower, even over a BATCH; anyway, it will update the internal Cache
- will be executed as a regular SQL statement:
  UPDATE table SET fieldname=fieldvalue WHERE RowID IN (...)  
- warning: this method will call directly EngineExecute(), and will work just fine with SQLite3, but some other DB engines may not allow a huge number of items within the IN(...) clause

function UpdateFieldIncrement(Table: TSQLRecordClass; ID: TID; const FieldName: RawUTF8; Increment: Int64=1): boolean; virtual;

_Increments one integer field value_
- if available, this method will use atomic value modification, e.g.
  UPDATE table SET field=field+?

procedure AdministrationExecute(const DatabaseName, SQL: RawUTF8; var result: TServiceCustomAnswer); virtual;

_Used e.g. by IAdministratedDaemon to implement "pseudo-SQL" commands_
- this default implementation will handle #time #model #rest commands
procedure AppendListAsJsonArray(Table: TSQLRecordClass; const FormatSQLWhere: RawUTF8; const BoundsSQLWhere: array of const; const OutputFieldName: RawUTF8; W: TJSONSerializer; const CustomFieldsCSV: RawUTF8='');

Get and append a list of members as an expanded JSON array
- implements REST GET collection
- generates '[[rec1],[rec2],...]' using a loop similar to:
  while FillOne do .. AppendJsonObject() ..
- for better server speed, the WHERE clause should use bound parameters identified as '?' in the FormatSQLWhere statement, which is expected to follow the order of values supplied in BoundsSQLWhere open array - use DateToSQL()/DateTimeToSQL() for TDateTime, or directly any integer, double, currency, RawUTF8 values to be bound to the request as parameters
- if OutputFieldName is set, the JSON array will be written as a JSON, property i.e. surrounded as "OutputFieldName":[...], - note ending ',
- CustomFieldsCSV can be the CSV list of field names to be retrieved
- if CustomFieldsCSV is "," will get all simple fields, excluding BLOBs
- if CustomFieldsCSV is ",", will get ALL fields, including ID and BLOBs
- is just a wrapper around TSQLRecord.AppendFillAsJsonArray()

procedure AsynchBatchRawAppend(Table: TSQLRecordClass; SentData: TTextWriter);

Append some JSON content in a BATCH to be written in a background thread
- could be used to emulate AsynchBatchAdd() with an already pre-computed JSON object, as stored in a TTextWriter instance
- is a wrapper around BackgroundTimer.AsynchBatchRawAppend()
- this method is thread-safe

procedure AsynchInterning(Interning: TRawUTF8Interning; InterningMaxRefCount: integer=2; PeriodMinutes: integer=5);

Allows background garbage collection of specified RawUTF8 interning
- will run Interning.Clean(2) every 5 minutes by default
- set InterningMaxRefCount=0 to disable process of the Interning instance
- note that InterningMaxRefCount and PeriodMinutes parameters (if not 0), are common for all TRawUTF8Interning instances (the latest value wins)
- you may e.g. run the following to clean up TDocVariant interned RawUTF8:
  aRest.AsynchInterning(DocVariantType.InternNames);
  aRest.AsynchInterning(DocVariantType.InternValues);

procedure AsynchRedirect(const aGUID: TGUID; const aDestinationInstance: TInterfacedObject; out aCallbackInterface; const aOnResult: TOnAsynchRedirectResult=nil); overload;

Define asynchronous execution of interface methods in a background thread
- this class allows to implements any interface via a fake class, which will redirect all methods calls into calls of another interface, but as a FIFO in a background thread, shared with TimerEnable/TimerDisable process
- it is an elegant resolution to the most difficult implementation problem of SOA callbacks, which is to avoid race condition on reentrance, e.g. if a callback is run from a thread, and then the callback code try to execute something in the context of the initial thread, protected by a critical section (mutex)
- is a wrapper around BackgroundTimer.AsynchRedirect()
procedure AsynchRedirect(const aGUID: TGUID; const aDestinationInterface: IInvokable; out aCallbackInterface; const aOnResult: TOnAsynchRedirectResult=nil); overload;

*Define asynchronous execution of interface methods in a background thread*
- this class allows to implements any interface via a fake class, which will redirect all methods calls into calls of another interface, but as a FIFO in a background thread, shared with TimerEnable/TimerDisable process
- it is an elegant resolution to the most difficult implementation problem of SOA callbacks, which is to avoid race condition on reentrance, e.g. if a callback is run from a thread, and then the callback code try to execute something in the context of the initial thread, protected by a critical section (mutex)
- is a wrapper around BackgroundTimer.AsynchRedirect()

procedure BeginCurrentThread(Sender: TThread); virtual;

*You can call this method in TThread.Execute to ensure that the thread will be taken into account during process*
- this abstract method won't do anything, but TSQLRestServer's will

procedure Commit(SessionID: cardinal; RaiseException: boolean=false); virtual;

*End a transaction*
- implements REST END collection
- write all pending SQL statements to the disk
- default implementation just reset the protected fTransactionActiveSession flag
- the supplied SessionID will allow multi-user transaction safety on the Server-Side: all database modification from another session will wait for the global transaction to be finished; on Client-side, the SessionID is just ignored (TSQLRestClient will override this method with a default SessionID=CONST_AUTHENTICATION_NOT_USED=1 parameter)
- if you have an external database engine which expect transactions to take place in the same thread, ensure TSQLRestServer force execution of this method when accessed from RESTful clients in the same thread, e.g.:
  
  ```pascal
  AcquireExecutionMode[execORMWrite] := amBackgroundThread;
  AcquireWriteMode := amBackgroundThread; // same as previous
  ```
- by default, any exception will be catch and ignored, unless RaiseException is set to TRUE so that the caller will be able to handle it

procedure DefinitionTo(Definition: TSynConnectionDefinition); virtual;

*Save the TSQLRest properties into a persistent storage object*
- you can then use TSQLRest.CreateFrom() to re-instantiate it
- current Definition.Key value will be used for the password encryption
- this default implementation will set the class name in Definition.Kind: inherited classes should override this method and serialize other properties, then override RegisteredClassCreateFrom() protected method to initiate the very same instance

procedure DefinitionToFile(const aJSONFile: TFileName; aKey: cardinal=0);

*Save the properties into a JSON file*
- you can then use TSQLRest.CreateFromFile() to re-instantiate it
- you can specify a custom Key, if the default is not enough for you
procedure EndCurrentThread(Sender: TThread); virtual;

You can call this method just before a thread is finished to ensure e.g. that the associated external DB connection will be released
- this abstract method will call fLogClass.Add.NotifyThreadEnded but TSQLRestServer.EndCurrentThread will do the main process

procedure QueryAddCustom(aTypeInfo: pointer; aEvent: TSQLQueryEvent; const aOperators: TSQLQueryOperators);

Add a custom query
- one event handler with an enumeration type containing all available query names
- and associated operators

procedure RollBack(SessionID: cardinal); virtual;

Abort a transaction
- implements REST ABORT collection
- restore the previous state of the database, before the call to TransactionBegin
- default implementation just reset the protected fTransactionActiveSession flag
- the supplied SessionID will allow multi-user transaction safety on the Server-Side: all database modification from another session will wait for the global transaction to be finished; on Client-side, the SessionID is just ignored (TSQLRestClient will override this method with a default SessionID=CONST_AUTHENTICATION_NOT_USED=1 parameter)
- if you have an external database engine which expect transactions to take place in the same thread, ensure TSQLRestServer force execution of this method when accessed from RESTful clients in the same thread, e.g.:

```pascal
AcquireExecutionMode[execORMWrite] := amBackgroundThread;
AcquireWriteMode := amBackgroundThread; // same as previous
```

procedure SetCustomEncryption(aes: TAESAbstract; sign: PSynSigner; comp: TAlgoCompress; const uriignore: RawUTF8='');

Initialize some custom AES encryption and/or digital signature, with optional compression
- will intercept the calls by setting OnDecryptBody/OnEncryptBody events
- will own the supplied aes instance or won’t encrypt the content if nil
- will digitally sign the content body and uri with the supplied TSynSigner, or won't compute any digital signature if sign=nil
- if both aes and sign are nil, then call interception is disabled
- you can optionally specify a compression algorithm (like AlgoSynLZ or AlgoDeflate/AlgoDeflateFast) to be applied before encryption
- any URI starting with urigignore characters won't be encrypted: it could be used to define a method-based service for handshake and aes/sign mutual agreement
- TSQLRestServer will require incoming requests to be of the corresponding [aesclass][signalgo]/[originaltype] HTTP content-type e.g. 'aesofb256sha256/application/json' - any plain request will be rejected
- note that it will only encrypt and sign the HTTP requests bodies, so URI or plain GET won't be checked - as such, it is not a replacement of TSQLRestServerAuthentication nor TWebSocketProtocolBinary encryption, but a cheap alternative to HTTPS, when you need to protect HTTP flow from MiM attacks (e.g. in a IoT context) with simple and proven algorithms

procedure WriteLock;

Enter the Mutex associated with the write operations of this instance
- just a wrapper around fAcquireExecution[execORMWrite].Safe.Lock
procedure WriteUnlock;

    Leave the Mutex associated with the write operations of this instance
    - just a wrapper around fAcquireExecution[execORMWrite].Safe.UnLock

property AcquireExecutionLockedTimeOut[Cmd: TSQLRestServerURIContextCommand]:
cardinal read GetAcquireExecutionLockedTimeOut write SetAcquireExecutionLockedTimeOut;

    The time (in milliseconds) to try locking internal commands of this class
    - this value is used only for AcquireExecutionMode[*]=amLocked
    - by default, TSQLRestServer.URI() will lock for Write ORM according to AcquireWriteTimeOut
      (i.e. AcquireExecutionLockedTimeOut[execORMWrite]) and other operations won't be locked
      nor have any time out set

property AcquireExecutionMode[Cmd: TSQLRestServerURIContextCommand]:
TSQLRestServerAcquireMode read GetAcquireExecutionMode write SetAcquireExecutionMode;

    How this class execute its internal commands
    - by default, TSQLRestServer.URI() will lock for Write ORM according to AcquireWriteMode (i.e.
      AcquireExecutionMode[execORMWrite]=amLocked) and other operations won't be protected
      (for better scaling)
    - you can tune this behavior by setting this property to the expected execution mode, e.g.
      execute all method-based services in a dedicated thread via
      aServer.AcquireExecutionMode[execSOAByMethod] := amBackgroundThread;
    - if you use external DB and a custom ConnectionTimeOutMinutes value, both read and write
      access should be locked, so you should set:
      aServer.AcquireExecutionMode[execORMGet] := am***;
      aServer.AcquireExecutionMode[execORMWrite] := am***;

    here, safe blocking am*** modes are any mode but amUnlocked, i.e. either amLocked,
    amBackgroundThread, amBackgroundORMSharedThread or amMainThread

property AcquireWriteMode: TSQLRestServerAcquireMode read GetAcquireExecutionMode write SetAcquireExecutionMode;

    How this class will handle write access to the database
    - is a common wrapper to AcquireExecutionMode[execORMWrite] property
    - default amLocked mode will wait up to AcquireWriteTimeOut milliseconds to have a single
      access to the server write ORM methods
    - amBackgroundThread will execute the write methods in a queue, in a dedicated unique thread
      (which can be convenient, especially for external database transaction process)
    - amBackgroundORMSharedThread will execute all ORM methods in a queue, in a dedicated
      unique thread, shared for both execORMWrite and execORMGet, but still dedicated for
      execSOAByMethod and execSOAByInterface
    - a slower alternative to amBackgroundThread may be amMainThread
    - you can set amUnlocked for a concurrent write access, but be aware that it may lead into
      multi-thread race condition issues, depending on the database engine used
property AcquireWriteTimeOut: cardinal index execORMWrite read GetAcquireExecutionLockedTimeOut write SetAcquireExecutionLockedTimeOut;

The time (in mill seconds) which the class will wait for acquiring a write access to the database, when AcquireWriteMode is amLocked
- is a common wrapper to AcquireExecutionLockedTimeOut[execORMWrite]
- in order to handle safe transactions and multi-thread safe writing, the server will identify transactions using the client Session ID: this property will set the time out wait period
- default value is 5000, i.e. TSQLRestServer.URI will wait up to 5 seconds in order to acquire the right to write on the database before returning a "408 Request Time-out" status error

property BackgroundTimer: TSQLRestBackgroundTimer read fBackgroundTimer;

Low-level background timer thread associated with this TSQLRest
- contains nil if TimerEnable/AsynchInvoke was never executed
- you may instantiate your own TSQLRestBackgroundTimer instances, if more than one working thread is needed

property Cache: TSQLRestCache read GetCache;

Access the internal caching parameters for a given TSQLRecord
- will always return a TSQLRestCache instance, creating one if needed
- purpose of this caching mechanism is to speed up retrieval of some common values at either Client or Server level (like configuration settings)
- by default, this CRUD level per-ID cache is disabled
- use Cache.SetCache() and Cache.SetTimeOut() methods to set the appropriate configuration for this particular TSQLRest instance
- only caching synchronization is about the direct RESTful/CRUD commands: RETRIEVE, ADD, UPDATE and DELETE (that is, a complex direct SQL UPDATE or via TSQLRecordMany pattern won't be taken into account - only exception is TSQLRestStorage tables accessed as SQLite3 virtual table)
- this caching will be located at the TSQLRest level, that is no automated synchronization is implemented between TSQLRestClient and TSQLRestServer - you shall ensure that your business logic is safe, calling Cache.Flush() overloaded methods on purpose: better no cache than unproper cache - "premature optimization is the root of all evil"

property LogClass: TSynLogClass read GetLogClass write SetLogClass;

The logging class used for this instance
- is set by default to SQLite3Log, but could be set to a custom class

property LogFamily: TSynLogFamily read fLogFamily;

The logging family used for this instance
- is set by default to SQLite3Log.Family, but could be set to something else by setting a custom class to the LogClass property

property Model: TSQLModel read fModel;

The Database Model associated with this REST Client or Server

property OnDecryptBody: TNotifyRestBody read fOnDecryptBody write fOnDecryptBody;
Event called before TSQLRestServer.URI or after TSQLRestClient.URI
- defined e.g. by SetCustomEncryption

property OnEncryptBody: TNotifyRestBody read fOnEncryptBody write fOnEncryptBody;
Event called after TSQLRestServer.URI or before TSQLRestClient.URI
- defined e.g. by SetCustomEncryption
### property PrivateGarbageCollector: TSynObjectList

**read** fPrivateGarbageCollector;

- A local "Garbage collector" list, for some classes instances which must live during the whole TSQLRestServer process
- is used internally by the class, but can be used for business code

### property ServerTimestamp: TTimeLog

**read** GetServerTimestamp
**write** SetServerTimestamp;

*The current UTC Date and Time, as retrieved from the server*
- this property will return the timestamp as TTimeLog / Int64 after correction from the Server returned time-stamp (if any)
- is used e.g. by TSQLRecord.ComputeFieldsBeforeWrite to update TModTime and TCreateTime published fields
- default implementation will return the executable UTC time, i.e. NowUTC so that any GUI code should convert this UTC value into local time
- on TSQLRestServer, if you use an external database, the TSQLDBConnection ServerTimestamp value will be set to this property
- you can use this value in a WHERE clause for a query, as such:
  
  ```pascal
  aRec.CreateAndFillPrepare(Client, 'Datum<=?', [TimeLogToSQL(Client.ServerTimestamp)]);
  ```
- or you could use ServerTimestamp everywhere in your code, when you need a reference time base

### property Services: TServiceContainer

**read** fServices;

*Access to the interface-based services list*
- may be nil if no service interface has been registered yet: so be aware that the following line may trigger an access violation if no ICalculator is defined on server side:
  
  ```pascal
  if fServer.Services['Calculator'].Get(Calc) then ...
  ```
- safer typical use, following the DI/IoC pattern, and which will not trigger any access violation if Services=nil, could be:
  
  ```pascal
  if fServer.Services.Resolve(ICalculator,Calc) then ...
  ```

### property ServicesRouting: TSQLRestServerURIContextClass

**read** fRoutingClass
**write** SetRoutingClass;

*The routing classes of the service remote request*
- by default, will use TSQLRestRoutingREST, i.e. an URI-based layout which is secure (since will use our RESTful authentication scheme), and also very fast
- but TSQLRestRoutingJSON_RPC can e.g. be set (on BOTH client and server sides), if the client will rather use JSON/RPC alternative pattern
- NEVER set the abstract TSQLRestServerURIContext class on this property

### TSQLRestThread = class(TThread)

*A simple TThread for doing some process within the context of a REST instance*
- also define a Start method for compatibility with older versions of Delphi
- inherited classes should override InternalExecute abstract method

**constructor** Create(aRest: TSQLRest; aOwnRest, aCreateSuspended: boolean);

*Initialize the thread*
- if aOwnRest is TRUE, the supplied REST instance will be owned by this thread
Synopses: mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

destructor Destroy; override;

  Finalize the thread
  - and the associated REST instance if OwnRest is TRUE

function SleepOrTerminated(MS: integer): boolean;
  Safe version of Sleep() which won't break the thread process
  - returns TRUE if the thread was Terminated
  - returns FALSE if successfully waited up to MS milliseconds

procedure Start;
  Method to be called to start the thread
  - Resume is deprecated in the newest RTL, since some OS - e.g. Linux - do not implement this
    pause/resume feature; we define here this method for older versions of Delphi

procedure Terminate; reintroduce;
  Reintroduced to call TerminatedSet

procedure TerminatedSet; virtual;
  Properly terminate the thread
  - called by reintroduced Terminate

procedure WaitForNotExecuting(maxMS: integer=500);
  Wait for Execute to be ended (i.e. fExecuting=false)

property Event: TEvent read fEvent;
  A event associated to this thread

property Executing: boolean read fExecuting;
  Publishes the thread executing state (set when Execute leaves)

property Log: TSynLog read fLog;
  Read-only access to the TSynLog instance of the associated REST instance

property OwnRest: boolean read fOwnRest;
  TRUE if the associated REST instance will be owned by this thread

property Rest: TSQLRest read FRest;
  Read-only access to the associated REST instance

property Safe: TSynLocker read fSafe;
  A critical section is associated to this thread
  - could be used to protect shared resources within the internal process

property Terminated;
  Publishes the thread running state

TSQLRestServerNamedPipe = class(TSQLRestThread)
  Server thread accepting connections from named pipes

  constructor Create(aServer: TSQLRestServer; const PipeName: TFileName);
  reintroduce;
    Create the server thread
**TSQLRestServerNamedPipeResponse** = class(TSQLRestThread)

Server child thread dealing with a connection through a named pipe

**constructor** Create(aServer: TSQLRestServer; aMasterThread: TSQLRestServerNamedPipe; aPipe: cardinal); reintroduce;

Create the child connection thread

**destructor** Destroy; override;

Release all associated memory, and decrement fMasterThread.fChildRunning

**TSQLRestServerURIPagingParameters** = record

Structure used to specify custom request paging parameters for TSQLRestServer
- default values are the one used for YUI component paging (i.e. PAGINGPARAMETERS_YAHOO constant, as set by TSQLRestServer.Create)
- warning: using paging can be VERY expensive on Server side, especially when used with external databases (since all data is retrieved before paging, when SQLite3 works in virtual mode)

Dir: RawUTF8;

Parameter name used to specify the request sort direction
- default value is 'DIR='

Results: RawUTF8;

Parameter name used to specify the request the page size (LIMIT clause)
- default value is 'RESULTS='

Select: RawUTF8;

Parameter name used to specify the request field names
- default value is 'SELECT='

SendTotalRowsCountFmt: RawUTF8;

Returned JSON field value of optional total row counts
- default value is nil, i.e. no total row counts field
- computing total row counts can be very expensive, depending on the database back-end used (especially for external databases)
- can be set e.g. to '"totalRows":%" value (note that the initial "," is expected by the produced JSON content, and % will be set with the value)
Sort: RawUTF8;

Parameter name used to specify the request sort order
- default value is 'SORT='

StartIndex: RawUTF8;

Parameter name used to specify the request starting offset
- default value is 'STARTINDEX='

Where: RawUTF8;

Parameter name used to specify the request WHERE clause
- default value is 'WHERE='

TSQLAuthGroup = class(TSQLRecord)

Table containing the available user access rights for authentication
- this class should be added to the TSQLModel, together with TSQLAuthUser, to allow authentication support
- you can inherit from it to add your custom properties to each user info: TSQLModel will search for any class inheriting from TSQLAuthGroup to manage per-group authorization data
- by default, it won't be accessible remotely by anyone

class procedure InitializeTable(Server: TSQLRestController; const FieldName: RawUTF8; Options: TSQLInitializeTableOptions); override;

Called when the associated table is created in the database
- on a new database, if TSQLAuthUser and TSQLAuthGroup tables are defined in the associated TSQLModel, it this will add 'Admin', 'Supervisor', and 'User' rows in the AuthUser table (with 'synopse' as default password), and associated 'Admin', 'Supervisor', 'User' and 'Guest' groups, with the following access rights to the AuthGroup table:

<table>
<thead>
<tr>
<th></th>
<th>POSTSQL</th>
<th>SELECTSQL</th>
<th>Service</th>
<th>AuthR</th>
<th>AuthW</th>
<th>TablesR</th>
<th>TablesW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admin</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Supervisor</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>User</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Guest</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

- 'Admin' will be the only able to execute remote not SELECT SQL statements for POST commands (reSQL flag in TSQLAccessRights.AllowRemoteExecute) and modify the Auth tables (i.e. AuthUser and AuthGroup)
- 'Admin' and 'Supervisor' will allow any SELECT SQL statements to be executed, even if the table can't be retrieved and checked (corresponding to the reSQLSelectWithoutTable flag)
- 'User' won't have the reSQLSelectWithoutTable flag, nor the right to retrieve the Auth tables data for other users
- 'Guest' won't have access to the interface-based remote JSON-RPC service (no reService flag), nor perform any modification to a table: in short, this is an ORM read-only limited user
- you MUST override the default 'synopse' password to a custom value, or at least customize the global AuthAdminDefaultPassword, AuthSupervisorDefaultPassword, AuthUserDefaultPassword variables
- of course, you can change and tune the settings of the AuthGroup and AuthUser tables, but only 'Admin' group users will be able to remotely modify the content of those two tables

property AccessRights: RawUTF8 index 1600 read fAccessRights write fAccessRights;

A textual representation of a TSQLAccessRights buffer
property Ident: RawUTF8  index 50  read fIdent  write fIdent  stored AS_UNIQUE;

The access right identifier, ready to be displayed
- the same identifier can be used only once (this column is marked as unique via a "stored
AS_UNIQUE" (i.e. "stored false") attribute)
- so you can retrieve a TSQLAuthGroup ID from its identifier, as such:

UserGroupID := fClient.MainFieldID(TSQLAuthGroup, 'User');

property SessionTimeout: integer  read fSessionTimeOut  write fSessionTimeOut;
The number of minutes a session is kept alive

property SQLAccessRights: TSQLAccessRights  read GetSQLAccessRights  write SetSQLAccessRights;
Corresponding TSQLAccessRights for this authentication group
- content is converted into/from text format via AccessRight DB property (so it will be not fixed
e.g. by the binary TSQLFieldTables layout, i.e. the MAX_SQLTABLES constant value)

TSQLAuthUser = class(TSQLRecord)

Table containing the Users registered for authentication
- this class should be added to the TSQLModel, together with TSQLAuthGroup, to allow
authentication support
- you can inherit from it to add your custom properties to each user info: TSQLModel will search
for any class inheriting from TSQLAuthUser to manage per-user authorization data
- by default, it won't be accessible remotely by anyone; to enhance security, you could use the
TSynValidatePassWord filter to this table

class function ComputeHashedPassword(const aPasswordPlain: RawUTF8; const aHashSalt: RawUTF8=''; aHashRound: integer=20000): RawUTF8; virtual;
Static function allowing to compute a hashed password
- as expected by this class
- defined as virtual so that you may use your own hashing class
- you may specify your own values in aHashSalt/aHashRound, to enable
PBKDF2_HMAC_SHA256() use instead of plain SHA256(): it will increase security on storage side
(reducing brute force attack via rainbow tables)

procedure SetPassword(const aPasswordPlain, aHashSalt: RawUTF8; aHashRound: integer=20000);
Set the PasswordHashHexa field from a plain password content and salt
- use this method to specify aHashSalt/aHashRound values, enabling PBKDF2_HMAC_SHA256() use instead of plain SHA256(): it will increase security on storage side (reducing brute force attack via rainbow tables)
- you may use an application specific fixed salt, and/or append the user LogonName to make the
challenge unique for each TSQLAuthUser
- the default aHashRound=20000 is slow but secure - since the hashing process is expected to be
done on client side, you may specify your own higher/slower value, depending on the security
level you expect
property Data: TSQLRawBlob read fData write fData;

Some custom data, associated to the User
- Server application may store here custom data
- its content is not used by the framework but 'may' be used by your application

property DisplayName: RawUTF8 index 50 read fDisplayName write fDisplayName;

The User Name, as may be displayed or printed

property GroupRights: TSQLAuthGroup read fGroup write fGroup;

The associated access rights of this user
- access rights are managed by group
- in TAuthSession.User instance, GroupRights property will contain a REAL TSQLAuthGroup
instance for fast retrieval in TSQLRestServer.URI
- note that 'Group' field name is not allowed by SQLite

property LogonName: RawUTF8 index 20 read fLogonName write fLogonName stored
AS UNIQUE;

The User identification Name, as entered at log-in
- the same identifier can be used only once (this column is marked as unique via a "stored
AS_UNIQUE" - i.e. "stored false" - attribute), and therefore indexed in the database (e.g. hashed
in TSQLRestStorageInMemory)

property PasswordHashHexa: RawUTF8 index 64 read fPasswordHashHexa write
fPasswordHashHexa;

The hexa encoded associated SHA-256 hash of the password
- see TSQLAuthUser.ComputeHashedPassword() or SetPassword() methods
- store the SHA-256 32 bytes as 64 hexa chars

property PasswordPlain: RawUTF8 write SetPasswordPlain;

Able to set the PasswordHashHexa field from a plain password content
- in fact, PasswordHashHexa := SHA256('salt'+PasswordPlain) in UTF-8
- use SetPassword() method if you want to customize the hash salt value and use the much safer
PBKDF2_HMAC_SHA256 algorithm

TAuthSession = class(TSynPersistent)
Class used to maintain in-memory sessions
- this is not a TSQLRecord table so won't be remotely accessible, for performance and security
reasons
- the User field is a true instance, copy of the corresponding database content (for better speed)
- you can inherit from this class, to add custom session process

constructor Create(aCtx: TSQLRestServerURIContext; aUser: TSQLAuthUser);
reintroduce; virtual;

Initialize a session instance with the supplied TSQLAuthUser instance
- this aUser instance will be handled by the class until Destroy
- raise an exception on any error
- on success, will also retrieve the aUser.Data BLOB field content

destructor Destroy; override;

Will release the User and User.GroupRights instances
property AccessRights: TSQLAccessRights read fAccessRights;
  *Copy of the associated user access rights*
  - extracted from User.TSQLAuthGroup.SQLAccessRights

property GroupID: TID read GetGroupID;
  *The associated Group ID, as in User.GroupRights.ID*

property ID: RawUTF8 read fID;
  *The session ID number, as text*

property IDCardinal: cardinal read fIDCardinal;
  *The session ID number, as numerical value*
  - never equals to 1 (CONST_AUTHENTICATION_NOT_USED, i.e. authentication mode is not
    enabled), nor 0 (CONST_AUTHENTICATION_SESSION_NOT_STARTED, i.e. session still in
    handshake phase)

property Interfaces: TSynMonitorInputOutputObjArray read fInterfaces;
  *Per-session statistics about interface-based services*
  - Interfaces[] follows TSQLRestServer.Services.fListInterfaceMethod[] array
  - is initialized and maintained only if mlSessions is defined in TSQLRestServer.StatLevels property

property Methods: TSynMonitorInputOutputObjArray read fMethods;
  *Per-session statistics about method-based services*
  - Methods[] follows TSQLRestServer.fPublishedMethod[] array
  - is initialized and maintained only if mlSessions is defined in TSQLRestServer.StatLevels property

property PrivateKey: RawUTF8 read fPrivateKey;
  *The hexadecimal private key as returned to the connected client as 'SessionID+PrivateKey'*

property RemoteIP: RawUTF8 read fRemoteIP;
  *The remote IP, if any*
  - is extracted from SentHeaders properties

property SentHeaders: RawUTF8 read fSentHeaders;
  *The transmitted HTTP headers, if any*
  - can contain e.g. 'RemoteIp: 127.0.0.1' or 'User-Agent: Mozilla/4.0'

property TimeoutShr10: cardinal read fTimeOutShr10;
  *The timestamp (in numbers of 1024 ms) until a session is kept alive*
  - extracted from User.TSQLAuthGroup.SessionTimeout
  - is used for fast comparison with GetTickCount64 shr 10

property TimeoutTix: cardinal read fTimeOutTix;
  *Set by the Access() method to the current GetTickCount64 shr 10 timestamp + TimeoutSecs*

property User: TSQLAuthUser read fUser;
  *The associated User*
  - this is a true TSQLAuthUser instance, and User.GroupRights will contain also a true
    TSQLAuthGroup instance

property UserID: TID read GetUserID;
  *The associated User ID, as in User.ID*
property UserName: RawUTF8 read GetUserName;

    The associated User Name, as in User.LogonName

TSQLRestServerAuthentication = class(TObject)

Abstract class used to implement server-side authentication in TSQLRestServer
- inherit from this class to implement expected authentication scheme

constructor Create(aServer: TSQLRestServer); virtual;

    Initialize the authentication method to a specified server
    - you can define several authentication schemes for the same server

function Auth(Ctxt: TSQLRestServerURIContext): boolean; virtual; abstract;

    Called by the Server to implement the Auth RESTful method
    - overridden method shall return TRUE if the request has been handled
    - returns FALSE to let the next registered TSQLRestServerAuthentication class to try implementing the content
    - Ctxt.Parameters has been tested to contain an UserName=... value
    - method execution is protected by TSQLRestServer.fSessions.Lock

class function ClientSetUser(Sender: TSQLRestClientURI; const aUserName, aPassword: RawUTF8; aPasswordKind: TSQLRestServerAuthenticationClientSetUserPassword=passClear; const aHashSalt: RawUTF8=''; aHashRound: integer=20000): boolean; virtual;

    Class method to be used on client side to create a remote session
    - call this method instead of TSQLRestClientURI.SetUser() if you need a custom authentication class
    - if saoUserByLogonOrID is defined in the server Options, aUserName may be a TSQLAuthUser.ID and not a TSQLAuthUser.LogonName
    - if passClear is used, you may specify aHashSalt and aHashRound, to enable PBKDF2_HMAC_SHA256() use instead of plain SHA256(), and increase security on storage side (reducing brute force attack via rainbow tables)
    - will call the ModelRoot/Auth service, i.e. call TSQLRestServer.Auth() published method to create a session for this user
    - returns true on success

function RetrieveSession(Ctxt: TSQLRestServerURIContext): TAuthSession; virtual; abstract;

    Called by the Server to check if the execution context match a session
    - returns a session instance corresponding to the remote request, and fill Ctxt.Session* members according to in-memory session information
    - returns nil if this remote request does not match this authentication
    - method execution should be protected by TSQLRestServer.fSessions.Lock

class procedure ClientSessionSign(Sender: TSQLRestClientURI; var Call: TSQLRestURIParams); virtual; abstract;

    Class method to be called on client side to sign an URI
    - used by TSQLRestClientURI.URI()
    - shall match the method as expected by RetrieveSession() virtual method
property Options: TSQLRestServerAuthenticationOptions read fOptions write fOptions;
   Allow to tune the authentication process
   - default value is [saoUserByLogonOrID]

TSQLRestServerAuthenticationURI = class(TSQLRestServerAuthentication)
   Weak authentication scheme using URL-level parameter

function RetrieveSession(Ctxt: TSQLRestServerURIContext): TAuthSession; override;
   Will check URI-level signature
   - retrieve the session ID from 'session_signature=...' parameter
   - method execution should be protected by TSQLRestServer.fsessions.Lock

class procedure ClientSessionSign(Sender: TSQLRestClientURI; var Call: TSQLRestURIParams); override;
   Class method to be called on client side to add the SessionID to the URI
   - append '&session_signature=SessionID' to the url

TSQLRestServerAuthenticationSignedURI =
class(TSQLRestServerAuthenticationURI)
   Secure authentication scheme using URL-level digital signature
   - default suaCRC32 format of session_signature is
     Hexa8(SessionID)+
     Hexa8(Timestamp)+
     Hexa8(crc32('SessionID+HexaSessionPrivateKey')+Sha256('salt'+PassWord)+
     Hexa8(Timestamp)+url))

constructor Create(aServer: TSQLRestServer); override;
   Initialize the authentication method to a specified server

function RetrieveSession(Cxt: TSQLRestServerURIContext): TAuthSession; override;
   Will check URI-level signature
   - check session_signature=... parameter to be a valid digital signature
   - method execution should be protected by TSQLRestServer.fsessions.Lock

class procedure ClientSessionSign(Sender: TSQLRestClientURI; var Call: TSQLRestURIParams); override;
   Class method to be called on client side to sign an URI
   - generate the digital signature as expected by overridden RetrieveSession()
   - timestamp resolution is about 256 ms in the current implementation

property Algorithm: TSQLRestServerAuthenticationSignedURIAlgo write SetAlgorithm;
   Customize the session_signature signing algorithm
   - you need to set this value on the server side only; those known algorithms will be recognized
   by TSQLRestClientURI on the client side during the session handshake, to select the matching
   ComputeSignature function
**property** ComputeSignature: TSQLRestServerAuthenticationSignedURIComputeSignature
```
read fComputeSignature
write fComputeSignature;
```

*Customize the session_signature signing algorithm with a specific function*
- the very same function should be set on TSQLRestClientURI
- to select a known hash algorithm, you may change the Algorithm property

**property** NoTimestampCoherencyCheck: Boolean
```
read fNoTimestampCoherencyCheck
write SetNoTimestampCoherencyCheck;
```

*Allow any order when creating sessions*
- by default, signed sessions are expected to be sequential, and new signed session signature
can't be older in time than the last one, with a tolerance of TimestampCoherencySeconds
- but if your client is asynchronous (e.g. for AJAX requests), session may be rejected due to the
delay involved on the client side: you can set this property to TRUE to enabled a weaker but
more tolerant behavior

```
(aServer.AuthenticationRegister(TSQLRestServerAuthenticationDefault) as TSQLRestServerAuthenticationSignedURI).NoTimestampCoherencyCheck := true;
```

**property** TimestampCoherencySeconds: cardinal
```
read fTimestampCoherencySeconds
write SetTimestampCoherencySeconds;
```

*Time tolerance in seconds for the signature timestamps coherency check*
- by default, signed sessions are expected to be sequential, and new signed session signature
can't be older in time than the last one, with a tolerance time defined by this property
- default value is 5 seconds, which cover most kind of clients (AJAX or WebSockets), even over a
slow Internet connection

```tSQLRestServerAuthenticationDefault =
class(TSQLRestServerAuthenticationSignedURI)
```

*MORMot secure RESTful authentication scheme*
- this method will use a password stored via safe SHA-256 hashing in the TSQLAuthUser ORM table
function Auth(Ctxt: TSQLRestServerURIContext): boolean; override;

Will try to handle the Auth RESTful method with mORMot authentication
- to be called in a two pass "challenging" algorithm:
  GET ModelRoot/auth?UserName=...
  -> returns an hexadecimal nonce contents (valid for 5 minutes)
  GET ModelRoot/auth?UserName=...&PassWord=...&ClientNonce=...
  -> if password is OK, will open the corresponding session
    and return 'SessionID+HexaSessionPrivateKey'

The Password parameter as sent for the 2nd request will be computed as
Sha256(ModelRoot+Nonce+ClientNonce+UserName+Sha256('salt'+PassWord))
- the returned HexaSessionPrivateKey content will identify the current user logged and its
  corresponding session (the same user may have several sessions opened at once, each with its
  own private key)
- then the private session key must be added to every query sent to the server as a
  session_signature=???? parameter, which will be computed as such:

were the session_signature= parameter will be computed as such:
  Hexa8(SessionID)+Hexa8(Timestamp)+
  Hexa8(crc32('SessionID+HexaSessionPrivateKey'+Sha256('salt'+PassWord)+
    Hexa8(Timestamp)+url))
  with url='ModelRoot/url?A=1&B=2'

this query authentication uses crc32 for hashing instead of SHA-256 in order to lower the
Server-side CPU consumption; the salted password (i.e. TSQLAuthUser.PasswordHashHexa) and
client-side Timestamp are inserted inside the session_signature calculation to prevent naive
man-in-the-middle attack (MITM)
- the session ID will be used to retrieve the rights associated with the user which opened the
  session via a successful call to the Auth service
- when you don't need the session any more (e.g. if the TSQLRestClientURI instance is
  destroyed), you can call the service as such:
  GET ModelRoot/auth?UserName=...&Session=...
  for a way of computing SHA-256 in JavaScript, see for instance
  @http://www.webtoolkit.info/javascript-sha256.html

TSQLRestServerAuthenticationNone = class(TSQLRestServerAuthenticationURI)

MORMot weak RESTful authentication scheme
- this method will authenticate with a given username, but no signature
- on client side, this scheme is not called by TSQLRestClientURI.SetUser() method - so you have to
  write:
  TSQLRestServerAuthenticationNone.ClientSetUser(Client,'User','');

function Auth(Ctxt: TSQLRestServerURIContext): boolean; override;

Will try to handle the Auth RESTful method with mORMot authentication
- to be called in a weak one pass request:
  GET ModelRoot/auth?UserName=...
  -> if the specified user name exists, will open the corresponding
    session and return 'SessionID+HexaSessionPrivateKey'
TSQLRestServerAuthenticationHttpAbstract =

class(TSQLRestServerAuthentication)

Abstract class for implementing HTTP authentication
- do not use this abstract class, but e.g. TSQLRestServerAuthenticationHttpBasic
- this class will transmit the session_signature as HTTP cookie, not at URI level, so is expected to be used only from browsers or old clients

class function ClientSetUser(Sender: TSQLRestClientURI; const aUserName, aPassword: RawUTF8; aPasswordKind: TSQLRestServerAuthenticationClientSetUserPassword=passClear; const aHashSalt: RawUTF8=''; aHashRound: integer=20000): boolean;

Class method to be used on client side to create a remote session
- call TSQLRestServerAuthenticationHttpBasic.ClientSetUser() instead of TSQLRestClientURI.SetUser(), and never the method of this abstract class
- needs the plain aPassword, so aPasswordKind should be passClear
- returns true on success

function RetrieveSession(Ctxt: TSQLRestServerURIContext): TAuthSession; override;

Will check the caller signature
- retrieve the session ID from "Cookie: mORMot_session_signature=..." HTTP header
- method execution should be protected by TSQLRestServer.fSessions.Lock

class procedure ClientSessionSign(Sender: TSQLRestClientURI; var Call: TSQLRestURIParams); override;

Class method to be called on client side to sign an URI in Auth Basic resolution is about 256 ms in the current implementation
- set "Cookie: mORMot_session_signature=..." HTTP header

class procedure ClientSetUserHttpOnly(Sender: TSQLRestClientURI; const aUserName, aPasswordClear: RawUTF8); virtual;

Class method to be used on client side to force the HTTP header for the corresponding HTTP authentication, without creating any remote session
- call virtual protected method ComputeAuthenticateHeader()
- here the password should be given as clear content
- potential use case is to use a mORMot client through a HTTPS proxy, e.g. with TSQLRestServerAuthenticationHttpBasic authentication
- then you can use TSQLRestServerAuthentication*.ClientSetUser() to define any another "mORMot only" authentication
- this method is also called by the ClientSetUser() method of this class for a full client + server authentication via HTTP TSQLRestServerAuthenticationHttp*.ClientSetUser()
TSQLRestServerAuthenticationHttpBasic =
class(TSQLRestServerAuthenticationHttpAbstract)

Authentication using HTTP Basic scheme
- this protocol send both name and password as clear (just base-64 encoded) so should only be
  used over SSL / HTTPS, or for compatibility reasons
- will rely on TSQLRestServerAuthenticationNone for authorization
- on client side, this scheme is not called by TSQLRestClientURI.SetUser() method - so you have to
  write:
  TSQLRestServerAuthenticationHttpBasic.ClientSetUser(Client, 'User', 'password');
- for a remote proxy-only authentication (without creating any mORMot session), you can write:
  TSQLRestServerAuthenticationHttpBasic.ClientSetUserHttpOnly(Client, 'proxyUser', 'proxyPass');

function Auth(Ctxt: TSQLRestServerURIContext): boolean; override;
  Handle the Auth RESTful method with HTTP Basic
  - will first return HTTP_UNAUTHORIZED (401), then expect user and password to be supplied as
    incoming "Authorization: Basic ...." headers

function RetrieveSession(Ctxt: TSQLRestServerURIContext): TAuthSession; override;
  Will check URI-level signature
  - retrieve the session ID from 'session_signature=...' parameter
  - will also check incoming "Authorization: Basic ...." HTTP header
  - method execution should be protected by TSQLRestServer.fSessions.Lock

TSQLRestServerAuthenticationSSPI =
class(TSQLRestServerAuthenticationSignedURI)

Authentication of the current logged user using Windows Security Support Provider Interface (SSPI)
  or GSSAPI library on Linux
- is able to authenticate the currently logged user on the client side, using either NTLM (Windows
  only) or Kerberos - it will allow to safely authenticate on a mORMot server without prompting the
  user to enter its password
- if ClientSetUser() receives aUserName as ",", aPassword should be either " if you expect NTLM
  authentication to take place, or contain the SPN registration (e.g.
  'mymormotservice/myservice.mydomain.tld') for Kerberos authentication
- if ClientSetUser() receives aUserName as 'DomainName\Username', then authentication will
  take place on the specified domain, with aPassword as plain password value

constructor Create(aServer: TSQLRestServer); override;
  Initialize the authentication method to a specified server

destructor Destroy; override;
  Finalize internal memory structures

function Auth(Ctxt: TSQLRestServerURIContext): boolean; override;
  Will try to handle the Auth RESTful method with Windows SSPI API
  - to be called in a two pass algorithm, used to cypher the password
  - the client-side logged user will be identified as valid, according to a Windows SSPI API secure
    challenge
TSQLHttpServerDefinition = class(TSynPersistentWithPassword)

Parameters supplied to publish a TSQLRestServer via HTTP
- used by the overloaded TSQLHttpServer.Create(TSQLHttpServerDefinition) constructor in mORMotHttpServer.pas, and also in dddInfraSettings.pas

property Authentication: TSQLHttpServerRestAuthentication read FAuthentication write FAuthentication;
Which authentication is expected to be published

property BindPort: RawByteString read FBindPort write FBindPort;
Defines the port to be used for REST publishing
- may include an optional IP address to bind, e.g. '127.0.0.1:8888'

property EnableCORS: RawUTF8 read FEnableCORS write FEnableCORS;
Allow Cross-origin resource sharing (CORS) access
- set this property to '*' if you want to be able to access the REST methods from an HTML5 application hosted in another location, or define a CSV white list of TMatch-compatible origins
- will set e.g. the following HTTP header:
  Access-Control-Allow-Origin: *

property Https: boolean read FHttps write FHttps;
Defines if https:// protocol should be used
- implemented only by http.sys server under Windows, not by socket servers

property HttpSysQueueName: SynUnicode read FHttpSysQueueName write FHttpSysQueueName;
The displayed name in the http.sys queue
- used only by http.sys server under Windows, not by socket-based servers

property RemoteIPHeader: RawUTF8 read fRemoteIPHeader write fRemoteIPHeader;
The value of a custom HTTP header containing the real client IP
- by default, the RemoteIP information will be retrieved from the socket layer - but if the server runs behind some proxy service, you should define here the HTTP header name which indicates the true remote client IP value, mostly as 'X-Real-IP' or 'X-Forwarded-For'

property ThreadCount: byte read fThreadCount write fThreadCount;
How many threads the thread pool associated with this HTTP server should create
- if set to 0, will use default value 32
- this parameter may be ignored depending on the actual HTTP server used, which may not have any thread pool

property WebSocketPassword: RawUTF8 read fPassWord write fPassWord;
If defined, this HTTP server will use WebSockets, and our secure encrypted binary protocol
- when stored in the settings JSON file, the password will be safely encrypted as defined by TSynPersistentWithPassword
- use the inherited PlainPassword property to set or read its value
TSynAuthenticationRest = class(TSynAuthenticationAbstract)

TSynAuthentication* class using TSQLAuthUser/TSQLAuthGroup for credentials
- could be used e.g. for SynDBRemote access in conjunction with mORMot

constructor Create(aServer: TSQLRestServer; const aAllowedGroups: array of integer); reintroduce;

Initialize the authentication scheme
- you can optionally set the groups allowing to use SynDBRemote - if none is specify, username/password is enough

class function ComputeHash(Token: Int64; const UserName,PassWord: RawUTF8): cardinal; override;

To be used to compute a Hash on the client side, for a given Token
- the password will be hashed as expected by the GetPassword() method

procedure RegisterAllowedGroups(const aAllowedGroups: array of integer);

Add some new groups to validate an user authentication

TSQLRecordModification = class(TSQLRecord)

Common ancestor for tracking TSQLRecord modifications
- e.g. TSQLRecordHistory and TSQLRecordVersion will inherit from this class to track TSQLRecord changes

function ModifiedID: TID;

Returns the modified record ID, as stored in ModifiedRecord

function ModifiedTable(Model: TSQLModel): TSQLRecordClass;

Returns the modified record table, as stored in ModifiedRecord

function ModifiedTableIndex: integer;

Returns the record table index in the TSQLModel, as stored in ModifiedRecord

property ModifiedRecord: TID read fModifiedRecord write fModifiedRecord;

Identifies the modified record
- ID and table index in TSQLModel is stored as one RecordRef integer
- you can use ModifiedTable/ModifiedID to retrieve the TSQLRecord item
- in case of the record deletion, all matching TSQLRecordHistory won't be touched by TSQLRestServer.AfterDeleteForceCoherency(): so this property is a plain TID/Int64, not a TRecordReference field

property Timestamp: TModTime read fTimestamp write fTimestamp;

When the modification was recorded
- even if in most cases, this timestamp may be synchronized over TSQLRest instances (thanks to TSQLRestClientURI.ServerTimestampSynchronize), it is not safe to use this field as absolute: you should rather rely on pure monotonic ID/RowID increasing values (see e.g. TSQLRecordVersion)
TSQLRecordHistory = class(TSQLRecordModification)

Common ancestor for tracking changes on TSQLRecord tables
- used by TSQLRestServer.TrackChanges() method for simple fields history
- TSQLRestServer.InternalUpdateEvent will use this table to store individual row changes as
  SentDataJSON, then will compress them in History BLOB
- note that any layout change of the tracked TSQLRecord table (e.g. adding a new property) will
  break the internal data format, so will void the table

constructor CreateHistory(aClient: TSQLRest; aTable: TSQLRecordClass; aID: TID);
  Load the change history of a given record
  - then you can use HistoryGetLast, HistoryCount or HistoryGet() to access all previous stored
    versions

destructor Destroy; override;
  Finalize any internal memory

function HistoryCount: integer;
  Returns how many revisions are stored in the History BLOB
  - HistoryOpen() or CreateHistory() should have been called before
  - this method will ignore any previous HistoryAdd() call

function HistoryGet( Index: integer; Rec: TSQLRecord): boolean; overload;
  Retrieve an historical version
  - HistoryOpen() or CreateHistory() should have been called before
  - this method will ignore any previous HistoryAdd() call
  - will fill all simple properties of the supplied TSQLRecord instance

function HistoryGet( Index: integer): TSQLRecord; overload;
  Retrieve an historical version
  - HistoryOpen() or CreateHistory() should have been called before
  - this method will ignore any previous HistoryAdd() call
  - will return either nil, or a TSQLRecord with all simple properties set

function HistoryGet( Index: integer; out Event: TSQLHistoryEvent; out Timestamp: TModTime; Rec: TSQLRecord): boolean; overload;
  Retrieve an historical version
  - HistoryOpen() or CreateHistory() should have been called before
  - this method will ignore any previous HistoryAdd() call
  - if Rec=nil, will only retrieve Event and Timestamp
  - if Rec is set, will fill all simple properties of this TSQLRecord

function HistoryGetLast(Rec: TSQLRecord): boolean; overload;
  Retrieve the latest stored historical version
  - HistoryOpen() or CreateHistory() should have been called before
  - this method will ignore any previous HistoryAdd() call
  - you should not have to use it, since a TSQLRest.Retrieve() is faster
function HistoryGetLast: TSQLRecord; overload;

Retrieve the latest stored historical version
- HistoryOpen() or CreateHistory() should have been called before, otherwise it will return nil
- this method will ignore any previous HistoryAdd() call
- you should not have to use it, since a TSQLRest.Retrieve() is faster

function HistoryOpen(Model: TSQLModel): boolean;

Prepare to access the History BLOB content
- ModifiedRecord should have been set to a proper value
- returns FALSE if the History BLOB is incorrect (e.g. TSQLRecord layout changed): caller shall flush all previous history

function HistorySave(Server: TSQLRestServer; LastRec: TSQLRecord=nil): boolean;

Update the History BLOB field content
- HistoryOpen() should have been called before using this method - CreateHistory() won’t allow history modification
- if HistoryAdd() has not been used, returns false
- ID field should have been set for proper persistence on Server
- otherwise compress the data into History BLOB, deleting the oldest versions if resulting size is bigger than expected, and returns true
- if Server is set, write save the History BLOB to database
- if Server and LastRec are set, its content will be compared with the current record in DB (via a Retrieve() call) and stored: it will allow to circumvent any issue about inconsistent use of tracking, e.g. if the database has been modified directly, by-passing the ORM

procedure HistoryAdd(Rec: TSQLRecord; Hist: TSQLRecordHistory);

Add a record content to the History BLOB
- HistoryOpen() should have been called before using this method - CreateHistory() won’t allow history modification
- use then HistorySave() to compress and replace the History field

class procedure InitializeTable(Server: TSQLRestServer; const FieldName: RawUTF8; Options: TSQLInitializeTableOptions); override;

Called when the associated table is created in the database
- create index on History(ModifiedRecord,Event) for process speed-up

property Event: TSQLHistoryEvent read fEvent write fEvent;

The kind of modification stored
- is heArchiveBlob when this record stores the compress BLOB in History
- otherwise, SentDataJSON may contain the latest values as JSON

property History: TSQLRawBlob read fHistory write fHistory;

After some events are written as individual SentData content, they will be gathered and compressed within one BLOB field
- use HistoryOpen/HistoryCount/HistoryGet to access the stored data after a call to CreateHistory() constructor
- as any BLOB field, this one won’t be retrieved by default: use explicitly TSQLRest.RetrieveBlobFields(aRecordHistory) to get it if you want to access it directly, and not via CreateHistory()
**property** SentDataJSON: RawUTF8  index 4000 read fSentData write fSentData;

*For heAdd/heUpdate, the data is stored as JSON*
- note that we defined a default maximum size of 4KB for this column, to avoid using a CLOB here - perhaps it may not be enough for huge records - feedback is welcome...

**TSQLRecordTableDeleted** = **class**(TSQLRecord)

ORM table used to store the deleted items of a versioned table
- the ID/RowID primary key of this table will be the version number (i.e. value computed by TSQLRestServer.InternalRecordVersionCompute), mapped with the corresponding 'TableIndex shl 58' (so that e.g. TSQLRestServer.RecordVersionSynchronizeToBatch() could easily ask for the deleted rows of a given table with a single WHERE clause on the ID/RowID)

**property** Deleted: Int64  read fDeleted write fDeleted;

*This Deleted published field will track the deleted row*
- defined as Int64 and not TID, to avoid the generation of the index on this column, which is not needed here (all requests are about ID/RowID)

**TSQLRestTempStorageItem** = **record**

Used to store an entry in the TSQLRestTempStorage class

ID: TID;

*The ID of this entry*
- after an AddCopy(ForceID=false), is a "fake" ID, which is > maxInt

Kind: TSQLRestTempStorageItemKind;

*What is stored in this entry*

Value: TSQLRecord;

*The stored item, either after adding or updating*
- equals nil if the item has been deleted

ValueFields: TSQLFieldBits;

*Identify the fields stored in the Value instance*
- e.g. an Update() - or even an Add() - may only have set only simple or specific fields

**TSQLRestTempStorage** = **class**(TSynPersistentLock)

Abstract class used for temporary in-memory storage of TSQLRecord
- purpose of this class is to gather write operations (Add/Update/Delete)
- inherited implementations may send all updates at once to a server (i.e. "asynchronous write"), or maintain a versioned image of the content
- all public methods (AddCopy/AddOwned/Update/Delete/FlushAsBatch) are thread-safe, protected by a mutex lock

**constructor** Create(aClass: TSQLRecordClass); reintroduce; virtual;

*Initialize the temporary storage for a given class*

**destructor** Destroy; override;

*Finalize this temporary storage instance*
Synopsys mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

```pascal
function AddCopy(Value: TSQLRecord; ForceID: boolean; const FieldNames: RawUTF8=''): TID; overload;
  Add a copy of a TSQLRecord to the internal storage list
  - if ForceID is true, Value.ID will be supplied with the ID to add
  - if ForceID is false, a "fake" ID is returned, which may be used later on for Update() calls
  WARNING: but this ID should not be stored as a cross reference in another record, since it is private to this storage; the definitive ID will be returned eventually after proper persistence (e.g. sent as TSQLRestBatch to a mORMot server)
  - FieldNames can be the CSV list of field names to be set
  - if FieldNames is ",", will set all simple fields, excluding BLOBs
  - if FieldNames is "*", will set ALL fields, including BLOBs
  - this method will clone the supplied Value, and make its own copy for its internal storage - consider use AddOwned() if the caller does not need to store the instance afterwards

function AddOwned(Value: TSQLRecord; ForceID: boolean; const Fields: TSQLFieldBits): TID; overload;
  Add and own a TSQLRecord in the internal storage list
  - if ForceID is true, Value.ID will be supplied with the ID to add
  - if ForceID is false, a "fake" ID is returned, which may be used later on for Update() calls
  WARNING: but this ID should not be stored as a cross reference in another record, since it is private to this storage; the definitive ID will be returned eventually after proper persistence (e.g. sent as TSQLRestBatch to a mORMot server)
  - this overloaded version expects the fields to be specified as bits
  - this method will store the supplied Value, and let its internal storage owns it and manage its lifetime - consider use AddCopy() if the caller does need to store this instance afterwards
  - returns 0 in case of error (e.g. ForceID and no or duplicated Value.ID)

function AddOwned(Value: TSQLRecord; ForceID: boolean; const FieldNames: RawUTF8=''): TID; overload;
  Add and own a TSQLRecord in the internal storage list
  - if ForceID is true, Value.ID will be supplied with the ID to add
  - if ForceID is false, a "fake" ID is returned, which may be used later on for Update() calls
  WARNING: but this ID should not be stored as a cross reference in another record, since it is private to this storage; the definitive ID will be returned eventually after proper persistence (e.g. sent as TSQLRestBatch to a mORMot server)
  - FieldNames can be the CSV list of field names to be set
  - if FieldNames is ",", will set all simple fields, excluding BLOBs
  - if FieldNames is "*", will set ALL fields, including BLOBs
  - this method will store the supplied Value, and let its internal storage owns it and manage its lifetime - consider use AddCopy() if the caller does need to store this instance afterwards
  - returns 0 in case of error (e.g. ForceID and no or duplicated Value.ID)

function FlushAsBatch(Rest: TSQLRest; AutomaticTransactionPerRow: cardinal=1000): TSQLRestBatch;
  Convert the internal list as a TSQLRestBatch instance, ready to be sent to the server

function FromEvent(Event: TSQLEvent; ID: TID; const JSON: RawUTF8): boolean;
  Add, update or delete a TSQLRecord in the internal storage list
  - could be used from a TNotifySQLEvent/InternalUpdateEvent(seAdd) callback
  - here the value to be added is supplied as a JSON object and a ID field
  - returns false in case of error (e.g. duplicated ID or void JSON)
```
function Update(Value: TSQLRecord; const Fields: TSQLFieldBits): boolean; overload;

Update a TSQLRecord and store the new values in the internal storage list
- Value.ID is used to identify the record to be updated (which may be a just added "fake" ID)
- this overloaded version expects the fields to be specified as bits
- the supplied Value won't be owned by this instance: the caller should release it when Value is no longer needed
- returns false in case of error (e.g. unknown ID or no field set)

function Update(Value: TSQLRecord; const FieldNames: RawUTF8=''): boolean; overload;

Update a TSQLRecord and store the new values in the internal storage list
- Value.ID is used to identify the record to be updated (which may be a just added "fake" ID)
- FieldNames can be the CSV list of field names to be updated
- if FieldNames is ",", will update all simple fields, excluding BLOBs
- if FieldNames is ",*", will update ALL fields, including BLOBs
- the supplied Value won't be owned by this instance: the caller should release it when Value is no longer needed
- returns false in case of error (e.g. unknow ID or invalid fields)

procedure Delete(const ID: TID);
Mark a TSQLRecord as deleted in the internal storage list

property Count: integer read fCount;
How many entries are stored in the low-level storage list

property Item: TSQLRestTempStorageItemDynArray read fItem;
Direct access to the low-level storage list
- the Count property is the number of items, length(Item) is the capacity
- the list is stored in increasing ID order

TSQLRestServerMonitor = class(TSynMonitorServer)
Used for high-level statistics in TSQLRestServer.URI()

constructor Create(aServer: TSQLRestServer); reintroduce;
No overridden Changed: TSQLRestServer.URI will do it in finally block initialize the instance

destructor Destroy; override;
Finalize the instance

function NotifyThreadCount(delta: integer): integer;
Update and returns the CurrentThreadCount property
- this method is thread-safe

procedure NotifyORM(aMethod: TSQLURIMethod);
Update the Created/Read/Updated/Deleted properties
- this method is thread-safe

procedure NotifyORMTable(TableIndex, DataSize: integer; Write: boolean; const MicroSecondsElapsed: QWord);
Update the per-table statistics
- this method is thread-safe
procedure ProcessSuccess(IsOutcomingFile: boolean); virtual;
  Should be called when a task successfully ended
  - thread-safe method

property Created: TSynMonitorCount64 read fCreated;
  How many Create/Add ORM operations did take place

property CurrentThreadCount: TSynMonitorOneCount read fCurrentThreadCount;
  Number of current declared thread counts
  - as registered by BeginCurrentThread/EndCurrentThread

property Deleted: TSynMonitorCount64 read fDeleted;
  How many Delete ORM operations did take place

property OutcomingFiles: TSynMonitorCount64 read fOutcomingFiles;
  Count of files transmitted directly (not part of Output size property)
  - i.e. when the service uses STATICFILE_CONTENT_TYPE/HTTP_RESP_STATICFILE as content type
to let the HTTP server directly serve the file content

property Read: TSynMonitorCount64 read fRead;
  How many Read/Get ORM operations did take place

property ServiceInterface: TSynMonitorCount64 read fServiceInterface;
  Count of the remote interface-based service calls

property ServiceMethod: TSynMonitorCount64 read fServiceMethod;
  Count of the remote method-based service calls

property StartDate: RawUTF8 read fStartDate;
  When this monitoring instance (therefore the server) was created

property Success: TSynMonitorCount64 read fSuccess;
  Number of valid responses
  - i.e. which returned status code 200/HTTP_SUCCESS or 201/HTTP_CREATED
  - any invalid request will increase the TSynMonitor.Errors property

property Updated: TSynMonitorCount64 read fUpdated;
  How many Update ORM operations did take place

TSQLMonitorUsage = class(TSQLRecordNoCaseExtended)
  ORM table used to store TSynMonitorUsage information in TSynMonitorUsageRest
  - the ID primary field is the TSynMonitorUsageID (accessible fromUsageId public property) shifted by 16 bits (by default) to include a TSynUniqueIdentifierProcess value

function UsageID(aProcessIDShift: integer=16): integer;
  Compute the corresponding 23 bit TSynMonitorUsageID.Value time slice
  - according to the stored Process field, after bit shift
  - allows a custom aProcessIDShift if it is not set as default 16 bits

property Comment: RawUTF8 read fComment write fComment;
  A custom text, which may be used e.g. by support or developers
**property** Gran: TSynMonitorUsageGranularity read fGran write fGran;
   The granularity of the statistics of this entry

**property** Info: variant read fInfo write fInfo;
   The actual statistics information, stored as a TDocVariant JSON object

**property** Process: Int64 read fProcess write fProcess;
   Identify which application is monitored
   - match the lower bits of each record ID
   - by default, is expected to be a TSynUniqueIdentifierProcess 16-bit value

TSynMonitorUsageRest = class(TSynMonitorUsage)
   Will store TSynMonitorUsage information in TSQLMonitorUsage ORM tables
   - TSQLRecord.ID will be the TSynMonitorUsageID shifted by ProcessIDShift bits

**constructor** Create(aStorage: TSQLRest; aProcessID: Int64; aStoredClass: TSQLMonitorUsageClass=nil; aProcessIDShift: integer=16); reintroduce; virtual;
   Initialize storage via ORM
   - if a 16-bit TSynUniqueIdentifierProcess is supplied, it will be used to identify the generating process by shifting TSynMonitorUsageID values by aProcessIDShift bits (default 16 but you may increase it up to 40 bits)
   - will use TSQLMonitorUsage table, unless another one is specified

**destructor** Destroy; override;
   Finalize the process, saving pending changes

**property** ProcessID: Int64 read fProcessID;
   How the information could be stored for several processes
   - e.g. when several SOA nodes gather monitoring information in a shared (MongoDB) database
   - is by default a TSynUniqueIdentifierProcess value, but may be any integer up to ProcessIDShift bits as set in Create()

**property** ProcessIDShift: integer read fProcessIDShift;
   How process ID are stored within the mORMot TSQLRecord.ID
   - equals 16 bits by default, to match TSynUniqueIdentifierProcess resolution

**property** SaveBatch: TSQLRestBatch read fSaveBatch write fSaveBatch;
   You can set an optional Batch instance to speed up DB writing
   - when calling the Modified() method

**property** StoredClass: TSQLMonitorUsageClass read fStoredClass;
   The actual ORM class used for persistence

TSQLRestServerURI = object(TObject)
   Used to access a TSQLRestServer from its TSQLRestServerURIString URI
   - URI format is 'address:port/root', and may be transmitted as TSQLRestServerURIString text instances

Address: RawUTF8;
   The TSQLRestServer IP Address or DNS name
Port: RawUTF8;
The TSQLRestServer IP port

Root: RawUTF8;
The TSQLRestServer model Root

function Equals(const other: TSQLRestServerURI): boolean;
Returns TRUE if all field values do match, case insensitively

property URI: TSQLRestServerURIString read GetURI write SetURI;
Property which allows to read or set the Address/Port/Root fields as one UTF-8 text field (i.e. a TSQLRestServerURIString instance)
- URI format is 'address:port/root', but port or root are optional

TServicesPublishedInterfaces = object(TObject)
Used to publish all Services supported by a TSQLRestServer instance
- as expected by TSQLRestServer.ServicesPublishedInterfaces
- can be serialized as a JSON object via RecordLoadJSON/RecordSaveJSON

Names: TRawUTF8DynArray;
The list of supported services names
- in fact this is the Interface name without the initial 'I', e.g. 'Calculator' for ICalculator

PublicURI: TSQLRestServerURI;
How this TSQLRestServer could be accessed

TServicesPublishedInterfacesList = class(TSynPersistentLock)
Used e.g. by TSQLRestServer to store a list of TServicesPublishedInterfaces

Count: Integer;
How many items are actually stored in List[]

List: TServicesPublishedInterfacesDynArray;
The internal list of published services
- the list is stored in-order, i.e. it will follow the RegisterFromJSON() execution order: the latest registrations will appear last

constructor Create(aTimeoutMS: integer); reintroduce; virtual;
initialize the storage
- an optional time out period, in milliseconds, may be defined - but the clients should ensure that RegisterFromClientJSON() is called in order to refresh the list (e.g. from _contract_ HTTP body)

function FindService(const aServiceName: RawUTF8): TSQLRestServerURIDynArray;
Search for the latest registrations of a service, by name
- will lookup for the Interface name without the initial 'I', e.g. 'Calculator' for ICalculator
- warning: research is case-sensitive
- if the service name has been registered several times, all registration will be returned, the latest in first position
function FindServiceAll(const aServiceName: RawUTF8): TSQLRestServerURIStringDynArray; overload;

Return all services URI by name, from the registration list, as URIs
- will lookup for the Interface name without the initial 'I', e.g. 'Calculator' for ICalculator
- warning: research is case-sensitive
- the returned string will contain all matching server URI, the latest registration being the first to appear, e.g.
  ["addresslast:port/root","addressprevious:port/root","addressfirst:port/root"]

function FindURI(const aPublicURI: TSQLRestServerURI): integer;
Search for a public URI in the registration list

function RegisterFromServer(Client: TSQLRestClientURI): boolean;
Set the list from a remote TSQLRestServer
- will call /root/Stat?findservice=* URI, then RegisterFromServerJSON()

procedure FindServiceAll(const aServiceName: RawUTF8; aWriter: TTextWriter); overload;

Return all services URI by name, from the registration list, as JSON
- will lookup for the Interface name without the initial 'I', e.g. 'Calculator' for ICalculator
- warning: research is case-sensitive
- the returned JSON array will contain all matching server URI, encoded as a TSQLRestServerURI JSON array, the latest registration being the first to appear, e.g.
  [{"Address":"addresslast","Port":"port","Root":"root"},...]
- if aServiceName='*', it will return ALL registration items, encoded as a TServicesPublishedInterfaces JSON array, e.g.
  [{"PublicURI":{"Address":"1.2.3.4","Port":"123","Root":"root"},"Names":["Calculator"],...}]

procedure RegisterFromClientJSON(var PublishedJson: RawUTF8);
Add the JSON serialized TServicesPublishedInterfaces to the list
- called by TSQLRestServerURIContext.InternalExecuteSOAByInterface when the client provides its own services as _contract_ HTTP body
- warning: supplied PublishedJson will be parsed in place, so modified

procedure RegisterFromServerJSON(var PublishedJson: RawUTF8);
Set the list from JSON serialized TServicesPublishedInterfacesDynArray
- may be used to duplicate the whole TSQLRestServer.AssociatedServices content, as returned from /root/Stat?findservice=*
- warning: supplied PublishedJson will be parsed in place, so modified

property TimeOut: integer read fTimeOut write fTimeOut;
The number of milliseconds after which an entry expires
- is 0 by default, meaning no expiration
- you can set it to a value so that any service URI registered with RegisterFromJSON() AFTER this property modification may expire
TSQLRestServer = class(TSQLRest)

A generic REPresentational State Transfer (REST) server
- descendent must implement the protected EngineList() Retrieve() Add() Update() Delete() methods
- automatic call of this methods by a generic URI() RESTful function
- any published method of descendants must match TSQLRestServerCallBack prototype, and is expected to be thread-safe

Used for DI-2.1.1 (page 2543), DI-2.1.1.1 (page 2543), DI-2.1.1.2.1 (page 2544), DI-2.1.1.2.2 (page 2544), DI-2.1.1.2.3 (page 2544), DI-2.1.1.2.4 (page 2545), DI-2.2.1 (page 2548).

InternalState: Cardinal;

This integer property is incremented by the database engine when any SQL statement changes the database contents (i.e. on any not SELECT statement)
- its value can be published to the client on every remote request
- it may be used by client to avoid retrieve data only if necessary
- if its value is 0, this feature is not activated on the server, and the client must ignore it and always retrieve the content

OnAfterURI: TNotifyAfterURI;

Event triggered when URI() finished to process a request
- the supplied Ctxt parameter will give access to the command which has been executed, e.g. via Ctxt.Call.OutStatus or Ctxt.MicroSecondsElapsed
- since this event will be executed by every TSQLRestServer.URI call, it should better not make any slow process (like writing to a remote DB)
- see also TSQLRest.OnDecryptBody/OnEncryptBody, which is common to the client side, so may be better to implement shared process (e.g. encryption)

OnAuthenticationFailed: TNotifyAuthenticationFailed;

This event handler will be executed when a session failed to initialize (DenyOfService attack?) or the request is not valid (ManInTheMiddle attack?)
- e.g. if the URI signature is invalid, or OnSessionCreate event handler aborted the session creation by returning TRUE (in this later case, the Session parameter is not nil)
- you can access the current execution context from the Ctxt parameter, e.g. to retrieve the caller's IP and ban aggressive users in Ctxt.RemoteIP or the text error message corresponding to Reason in Ctxt.CustomErrorMsg

OnAuthenticationUserRetrieve: TOnAuthenticationUserRetrieve;

A custom method to retrieve the TSQLAuthUser instance for authentication
- will be called by TSQLRestServerAuthentication.GetUser() instead of plain SQLAuthUserClass.Create()
### OnBeforeURI: TNotifyBeforeURI;

*Event triggered when URI() starts to process a request*
- the supplied Ctxt parameter will give access to the command about to be executed, e.g.
- Ctxt.Command=execSOAByInterface will identify a SOA service execution, with the corresponding Service and ServiceMethodIndex parameters as set by TSQLRestServerURIContext.URIDecodeSOAByInterface
- should return TRUE if the method can be executed
- should return FALSE if the method should not be executed, and the callback should set the corresponding error to the supplied context e.g.
  ```pascal
  Ctxt.Error('Unauthorized method', HTTP_NOT_ALLOWED);
  ```
- since this event will be executed by every TSQLRestServer.URI call, it should better not make any slow process (like writing to a remote DB)
- see also TSQLRest.OnDecryptBody, which is common to the client side, so may be a better place for implementing shared process (e.g. encryption)

### OnBlobUpdateEvent: TNotifyFieldSQLEvent;

*A method can be specified here to trigger events after any blob update*
- is called AFTER update of one or several blobs, never on delete nor insert
- to be used only server-side, not to synchronize some clients: the framework is designed around a stateless RESTful architecture (like HTTP/1.1), in which clients ask the server for refresh (see TSQLRestClientURI.UpdateFromServer)

### OnErrorURI: TNotifyErrorURI;

*Event triggered when URI() failed to process a request*
- if Ctxt.ExecuteCommand raised an exception, this callback will be run with all needed information
- should return TRUE to execute Ctxt.Error(E,...), FALSE if returned content has already been set as expected by the client

### OnIdle: TNotifyEvent;

*Event triggered when URI() is called, and at least 128 ms is elapsed*
- could be used to execute some additional process after a period of time
- note that if TSQLRestServer.URI is not called by any client, this callback won’t be executed either

### OnInternalInfo: TOnInternalInfo;

*Event to customize the information returned by root/timestamp/info*
- called by TSQLRestServer.InternalInfo method
- you can add some application-level information for monitoring

### OnNotifyCallback: TSQLRestServerNotifyCallback;

*This event will be executed to push notifications from the server to a remote client, using a (fake) interface parameter*
- is nil by default, but may point e.g. to TSQLHttpServer.NotifyCallback
OnServiceCreateInstance: TOnServiceCreateInstance;

_This event will be executed by TServiceFactoryServer.CreateInstance_
- you may set a callback to customize a server-side service instance, i.e. inject class-level dependencies:

_procedure TMyClass.OnCreateInstance(
  Sender: TServiceFactoryServer; Instance: TInterfacedObject);
begin
  if Sender.ImplementationClass=TLegacyStockQuery then
    TLegacyStockQuery(Instance).fDbConnection := fDbConnection;
end;
- consider using a TInjectableObjectClass implementation for pure IoC/DI

OnSessionClosed: TNotifySQLSession;
_A method can be specified to be notified when a session is closed_
- for OnSessionClosed, the returning boolean value is ignored
- Ctxt is nil if the session is closed due to a timeout
- Ctxt is not nil if the session is closed explicitly by the client

OnSessionCreate: TNotifySQLSession;
_A method can be specified to be notified when a session is created_
- for OnSessionCreate, returning TRUE will abort the session creation - and you can set Ctxt.Call^.OutStatus to a corresponding error code
- it could be used e.g. to limit the number of client sessions

OnUpdateEvent: TNotifySQLEvent;
_A method can be specified here to trigger events after any table update_
- is called BEFORE deletion, and AFTER insertion or update
- note that the aSentData parameter does not contain all record fields, but only transmitted information: e.g. if only one field is updated, only this single field (and the ID) is available
- to be used only server-side, not to synchronize some clients: the framework is designed around a stateless RESTful architecture (like HTTP/1.1), in which clients ask the server for refresh (see TSQLRestClientURI.UpdateFromServer)

URIPagingParameters: TSQLRestServerURIPagingParameters;
_This property can be used to specify the URI parameters to be used for query paging_
- is set by default to PAGINGPARAMETERS_YAHOO constant by TSQLRestServer.Create() constructor

constructor Create(aModel: TSQLModel; aHandleUserAuthentication: boolean=false);
reintroduce; virtual;
_Server initialization with a specified Database Model_
- if HandleUserAuthentication is false, will set URI access rights to 'Supervisor' (i.e. all R/W access) by default
- if HandleUserAuthentication is true, will add TSQLAuthUser and TSQLAuthGroup to the TSQLModel (if not already there)

constructor CreateWithOwnModel(const Tables: array of TSQLRecordClass;
aHandleUserAuthentication: boolean=false; const aRoot: RawUTF8='root');
_Server initialization with a temporary Database Model_
- a Model will be created with supplied tables, and owned by the server
- if you instantiate a TSQLRestServerFullMemory or TSQLRestServerDB with this constructor, an in-memory engine will be created, with enough abilities to run regression tests, for instance
**Destructor**  Destroy; override;

Release memory and any existing pipe initialized by ExportServer()

**Function**  AfterDeleteForceCoherency(aTableIndex: integer; aID: TID): boolean; virtual;

This method is called internally after any successful deletion to ensure relational database coherency.

- reset all matching TRecordReference properties in the database Model, for database coherency, into 0
- delete all records containing a matched TRecordReferenceToBeDeleted property value in the database Model (e.g. TSQLRecordHistory)
- reset all matching TSQLRecord properties in the database Model, for database coherency, into 0
- important notice: we don't use FOREIGN KEY constraints in this framework, and handle all integrity check within this method (it's therefore less error-prone, and more cross-database engine compatible)

**Function**  AuthenticationRegister( aMethod: TSQLRestServerAuthenticationClass): TSQLRestServerAuthentication; overload;

Call this method to add an authentication method to the server.

- will return the just created TSQLRestServerAuthentication instance, or the existing instance if it has already been registered
- you can use this method to tune the authentication, e.g. if you have troubles with AJAX asynchronous callbacks:

```pascal
(aServer.AuthenticationRegister(TSQLRestServerAuthenticationDefault) as TSQLRestServerAuthenticationDefault).NoTimestampCoherencyCheck := true;
```

or if you want to customize the session_signature parameter algorithm:

```pascal
(aServer.AuthenticationRegister(TSQLRestServerAuthenticationDefault) as TSQLRestServerAuthenticationDefault).Algorithm := suaMD5;
```

**Function**  BanIP(const aIP: RawUTF8; aRemoveBan: boolean=false): boolean;

(un)register a banned IPv4 value

- any connection attempt from this IP Address will be rejected by

**Function**  CloseServerMessage: boolean;

End any currently initialized message-oriented server

**Function**  CloseServerNamedPipe: boolean;

End any currently initialized named pipe server

**Class Function**  CreateInMemoryForAllVirtualTables(aModel: TSQLModel; aHandleUserAuthentication: boolean): TSQLRestServer;

Create a new minimal TSQLRestServer instance, to be used with external SQL or NoSQL storage.

- will try to instantiate an in-memory TSQLRestServerDB, and if mORMotSQLite3.pas is not linked, fallback to a TSQLRestServerFullMemory
- used e.g. by TSQLRestMongoDBCreate() and TSQLRestExternalDBCreate()

**Function**  CreateSQLIndex(Table: TSQLRecordClass; const FieldName: RawUTF8; Unique: boolean; const IndexName: RawUTF8=''): boolean; overload;

Create an index for the specific FieldName

- will call CreateSQLMultiIndex() internally
function CreateSQLIndex(Table: TSQLRecordClass; const FieldNames: array of RawUTF8; Unique: boolean): boolean; overload;

Create one or multiple index(es) for the specific FieldName(s)

function CreateSQLMultiIndex(Table: TSQLRecordClass; const FieldNames: array of RawUTF8; Unique: boolean; IndexName: RawUTF8=''): boolean; virtual;

Create one index for all specific FieldName(s) at once
- will call any static engine for the index creation of such tables, or execute a CREATE INDEX IF NOT EXISTS on the main engine
- note that with SQLite3, your database schema should never contain two indices where one index is a prefix of the other, e.g. if you defined:
  aServer.CreateSQLMultiIndex(TEmails, ['Email', 'GroupID'], True);

Then the following index is not mandatory for SQLite3:
  aServer.CreateSQLIndex(TEmails, 'Email', False);

see "1.6 Multi-Column Indices" in @http://www.sqlite.org/queryplanner.html

function Delete(Table: TSQLRecordClass; const SQLWhere: RawUTF8): boolean;

override;

Implement Server-Side TSQLRest deletion with a WHERE clause
- will process all ORM-level validation, coherency checking and notifications together with a low-level SQL deletion work (if possible)

function Delete(Table: TSQLRecordClass; ID: TID): boolean; override;

Implement Server-Side TSQLRest deletion
- uses internally EngineDelete() function for calling the database engine
- call corresponding fStaticData[] if necessary
- this record is also erased in all available TRecordReference properties in the database Model, for relational database coherency

function ExportedAsMessageOrNamedPipe: Boolean;

Returns TRUE if remote connection is possible via named pipes or Windows messages

function ExportServer: boolean; overload;

Grant access to this database content from a dll using the global URIRequest() function
- returns true if the URIRequest() function is set to this TSQLRestServer
- returns false if a TSQLRestServer was already exported
- client must release all memory acquired by URIRequest() with GlobalFree()
function ExportServerMessage(const ServerWindowName: string): boolean;

Declare the server on the local machine to be accessible for local client connection, by using Windows messages
- the data is sent and received by using the standard and fast WM_COPYDATA message
- Windows messages are very fast (faster than named pipe and much faster than HTTP), but only work locally on the same computer
- create a new Window Class with the supplied class name (UnicodeString since Delphi 2009 for direct use of Wide Win32 API), and instantiate a window which will handle pending WM_COPYDATA messages
- the main server instance has to process the windows messages regularly (e.g. with Application.ProcessMessages)
- ServerWindowName ('DBSERVER' e.g.) will be used to create a Window name identifier
- allows only one ExportServer*() by running process
- returns true on success, false otherwise (ServerWindowName already used?)

Used for DI-2.1.1.2.3 (page 2544).

function ExportServerNamedPipe(const ServerApplicationName: TFileName): boolean;

Declare the server on the local machine as a Named Pipe: allows TSQLRestClientURNamedPipe local or remote client connection
- ServerApplicationName ('DBSERVER' e.g.) will be used to create a named pipe server identifier, it is of UnicodeString type since Delphi 2009 (use of Unicode FileOpen() version)
- this server identifier is appended to '\pipe\mORMot_' to obtain the full pipe name to initiate ('\pipe\mORMot_DBSERVER' e.g.)
- this server identifier may also contain a fully qualified path ('\pipe\ApplicationName' e.g.)
- allows only one ExportServer*() by running process
- returns true on success, false otherwise (ServerApplicationName already used?)

Used for DI-2.1.1.2.2 (page 2544).

function InternalUpdateEvent(aEvent: TSQLEvent; aTableIndex: integer; aID: TID; const aSentData: RawUTF8; aIsBlobFields: PSQLFieldBits): boolean; virtual;

Virtual method called when a record is updated
- default implementation will call the OnUpdateEvent/OnBlobUpdateEvent methods, if defined
- will also handle TSQLErrorHistory tables, as defined by TrackChanges()
- returns true on success, false if an error occurred (but action must continue)
- you can override this method to implement a server-wide notification, but be aware it may be the first step to break the stateless architecture of the framework

function InternalUpdateEventNeeded(aTableIndex: integer): boolean;

Check if OnUpdateEvent or change tracked has been defined for this table
- is used internally e.g. by TSQLRestServerDB.MainEngineUpdateField to ensure that the updated ID fields will be computed as expected

function JWTForUnauthenticatedRequestWhiteIP(const aIP: RawUTF8; aRemoveWhite: boolean=false): boolean;

(un)register a an IPv4 value to the JWT white list
- by default, a JWT validated by JWTForUnauthenticatedRequest will be accepted
- to avoid MiM (Man-In-the-Middle) attacks, if a JWT white list is defined using this method, any connection from a non registered IP will be rejected, even with a valid JWT
- WebSockets connections are secure enough to bypass this list
function MemberExists(Table: TSQLRecordClass; ID: TID): boolean; override;
  Overridden method for direct static class call (if any)

function RecordVersionCompute: TRecordVersion;
  Will compute the next monotonic value for a TRecordVersion field

function RecordVersionCurrent: TRecordVersion;
  Read only access to the current monotonic value for a TRecordVersion field

function RecordVersionSynchronizeMasterStart(ByPassAuthentication: boolean=false): boolean;
  Initiate asynchronous master/slave replication on a master TSQLRest
  - allow synchronization of a TSQLRecord table, using its TRecordVersion field, for real-time
    master/slave replication on the master side
  - this method will register the IServiceRecordVersion service on the server side, so that
    RecordVersionSynchronizeStartSlave() will be able to receive push notifications of any updates
  - this method expects the communication channel to be bidirectional, e.g. a
    mORMotHTTPServer's TSQLHttpServer in useBidirSocket mode

function RecordVersionSynchronizeSlave(Table: TSQLRecordClass; Master: TSQLRest;
  ChunkRowLimit: integer=0; OnWrite: TOnBatchWrite=nil): TRecordVersion;
  Synchronous master/slave replication from a slave TSQLRest
  - apply all the updates from another (distant) master TSQLRest for a given TSQLRecord table,
    using its TRecordVersion field, to the calling slave
  - both remote Master and local slave TSQLRestServer should have the supplied Table class in
    their data model (maybe in diverse order)
  - by default, all pending updates are retrieved, but you can define a value to ChunkRowLimit, so
    that the updates will be retrieved by smaller chunks
  - returns -1 on error, or the latest applied revision number (which may be 0 if there is no data in
    the table)
  - this method will use regular REST ORM commands, so will work with any communication
    channels: for real-time push synchronization, consider using
    RecordVersionSynchronizeMasterStart and RecordVersionSynchronizeSlaveStart over a
    bidirectional communication channel like WebSockets
  - you can use RecordVersionSynchronizeSlaveToBatch if your purpose is to access the updates
    before applying to the current slave storage

function RecordVersionSynchronizeSlaveStart(Table: TSQLRecordClass;
  MasterRemoteAccess: TSQLRestClientURI; OnNotify: TOnBatchWrite=nil): boolean;
  Initiate asynchronous master/slave replication on a slave TSQLRest
  - start synchronization of a TSQLRecord table, using its TRecordVersion field, for real-time
    master/slave replication on the slave side
  - this method will first retrieve any pending modification by regular REST calls to
    RecordVersionSynchronizeSlave, then create and register a callback instance using
    RecordVersionSynchronizeSubscribeMaster()
  - this method expects the communication channel to be bidirectional, e.g. a
    TSQLHttpClientWebsockets
  - the modifications will be pushed by the master, then applied to the slave storage, until
    RecordVersionSynchronizeSlaveStop method is called
  - an optional OnNotify event may be defined, which will be triggered for all incoming change,
    supplying the updated TSQLRecord instance
function RecordVersionSynchronizeSlaveStop(Table: TSQLRecordClass): boolean;

Finalize asynchronous master/slave replication on a slave TSQLRest
- stop synchronization of a TSQLRecord table, using its TRecordVersion field, for real-time master/slave replication on the slave side
- expect a previous call to RecordVersionSynchronizeSlaveStart

function RecordVersionSynchronizeSlaveToBatch(Table: TSQLRecordClass; Master: TSQLRest; var RecordVersion: TRecordVersion; MaxRowLimit: integer=0; OnWrite: TOnBatchWrite=nil): TSQLRestBatch;
virtual;

Synchronous master/slave replication from a slave TSQLRest into a Batch
- will retrieve all the updates from a (distant) master TSQLRest for a given TSQLRecord table, using its TRecordVersion field, and a supplied TRecordVersion monotonic value, into a TSQLRestBatch instance
- both remote Source and local TSQLRestServer should have the supplied Table class in each of their data model
- by default, all pending updates are retrieved, but you can define a value to MaxRowLimit, so that the updates will be retrieved by smaller chunks
- returns nil if nothing new was found, or a TSQLRestBatch instance containing all modifications since RecordVersion revision
- when executing the returned TSQLRestBatch on the database, you should set TSQLRestServer.RecordVersionDeleteIgnore := true so that the TRecordVersion fields will be forced from the supplied value
- usually, you should not need to use this method, but rather the more straightforward RecordVersionSynchronizeSlave()

function RecordVersionSynchronizeSubscribeMaster(Table: TSQLRecordClass; RecordVersion: TRecordVersion; const SlaveCallback: IServiceRecordVersionCallback): boolean; overload;

Low-level callback registration for asynchronous master/slave replication
- you should not have to use this method, but rather RecordVersionSynchronizeMasterStart and RecordVersionSynchronizeSlaveStart RecordVersionSynchronizeSlaveStop methods
- register a callback interface on the master side, which will be called each time a write operation is performed on a given TSQLRecord with a TRecordVersion field
- the callback parameter could be a TServiceRecordVersionCallback instance, which will perform all update operations as expected
- the callback process will be blocking for the ORM write point of view: so it should be as fast as possible, or asynchronous - note that regular callbacks using WebSockets, as implemented by SynBidirSock.pas and mORMotHTTPServer's TSQLHttpServer in useBidirSocket mode, are asynchronous
- if the supplied RecordVersion is not the latest on the server side, this method will return FALSE and the caller should synchronize again via RecordVersionSynchronize() to avoid any missing update
- if the supplied RecordVersion is the latest on the server side, this method will return TRUE and put the Callback notification in place
function RemoteDataCreate(aClass: TSQLRecordClass; aRemoteRest: TSQLRest): TSQLRestStorageRemote; virtual;

Create an external static redirection for a specific class
- call it just after Create, before TSQLRestServerDB.CreateMissingTables; warning: if you don't call this method before CreateMissingTable method is called, the table will be created as a regular table by the main database engine, and won't be static
- the specified TSQLRecord will have all its CRUD / ORM methods be redirected to aRemoteRest, which may be a TSQLRestClient or another TSQLRestServer instance (e.g. a fast SQLITE_MEMORY_DATABASE_NAME)
- if aRemoteRest is a TSQLRestClient, it should have been authenticated to the remote TSQLRestServer, so that CRUD / ORM operations will pass
- this will enable easy creation of proxies, or local servers, with they own cache and data model - e.g. a branch office server which may serve its local clients over Ethernet, but communicating to a main mORMot server via Internet, storing the corporate data in the main office server
- you may also share some tables (e.g. TSQLAuthUser and TSQLAuthGroup) between TSQLRestServer instances in a single service

function RetrieveBlobFields(Value: TSQLRecord): boolean; override;

Get all BLOB fields of the supplied value from the remote server
- this overridden method will execute the direct static class, if any

function ServiceContainer: TServiceContainer; override;

Access or initialize the internal IoC resolver, used for interface-based remote services, and more generally any Services.Resolve() call
- create and initialize the internal TServiceContainerServer if no service interface has been registered yet
- may be used to inject some dependencies, which are not interface-based remote services, but internal IoC, without the ServiceRegister() or ServiceDefine() methods - e.g.
  aRest.ServiceContainer.InjectResolver([TInfraRepoUserFactory.Create(aRest),true]);
- this overridden method will return a TServiceContainerServer instance
- you may enable SOA audit trail for all methods execution:
  (aRestSOAServer.ServiceContainer as TServiceContainerServer).SetServiceLog(aRestLogServer,TSQLRecordServiceLog);

function ServiceDefine(aImplementationClass: TInterfacedClass; const aInterfaces: array of TGUID; aInstanceCreation: TServiceInstanceImplementation=sicSingle; const aContractExpected: RawUTF8=''): TServiceFactoryServer; overload;

Register a Service class on the server side
- this method expects the interface(s) to have been registered previously:
  TInterfaceFactory.RegisterInterfaces([TypeInfo(IMyInterface),...]);

function ServiceDefine(aSharedImplementation: TInterfacedObject; const aInterfaces: array of TGUID; const aContractExpected: RawUTF8=''): TServiceFactoryServer; overload;

Register a Service instance on the server side
- this method expects the interface(s) to have been registered previously:
  TInterfaceFactory.RegisterInterfaces([TypeInfo(IMyInterface),...]);
- the supplied aSharedImplementation will be owned by this Server instance
function ServiceDefine(aClient: TSQLRest; const aInterfaces: array of TGUID; aInstanceCreation: TServiceInstanceImplementation=sicSingle; const aContractExpected: RawUTF8=''): boolean; overload;

Register a remote Service via its interface
- this method expects the interface(s) to have been registered previously:
  TInterfaceFactory.RegisterInterfaces([TypeInfo(IMyInterface),...]);

function ServiceMethodByPassAuthentication(const aMethodName: RawUTF8): integer;

Call this method to disable Authentication method check for a given published method-based service name
- by default, only Auth and Timestamp methods do not require the RESTful authentication of the URI; you may call this method to add another method to the list (e.g. for returning some HTML content from a public URI)
- if the supplied aMethodName='', all method-based services will bypass the authentication process
- returns the method index number

function ServiceRegister(aImplementationClass: TInterfacedClass; const aInterfaces: array of PTypeInfo; aInstanceCreation: TServiceInstanceImplementation=sicSingle; const aContractExpected: RawUTF8=''): TServiceFactoryServer; overload; virtual;

Register a Service class on the server side
- this methods expects a class to be supplied, and the exact list of interfaces to be registered to the server (e.g. [TypeInfo(IMyInterface)]) and implemented by this class
- class can be any TInterfacedObject, but TInterfacedObjectWithCustomCreate can be used if you need an overridden constructor
- instance implementation pattern will be set by the appropriate parameter
- will return the first of the registered TServiceFactoryServer created on success (i.e. the one corresponding to the first item of the aInterfaces array), or nil if registration failed (e.g. if any of the supplied interfaces is not implemented by the given class)
- you can use the returned TServiceFactoryServer instance to set the expected security parameters associated with this interface
- the same implementation class can be used to handle several interfaces (just as Delphi allows to do natively)

function ServiceRegister(aSharedImplementation: TInterfacedObject; const aInterfaces: array of PTypeInfo; const aContractExpected: RawUTF8=''): TServiceFactoryServer; overload; virtual;

Register a Service instance on the server side
- this methods expects a class instance to be supplied, and the exact list of interfaces to be registered to the server (e.g. [TypeInfo(IMyInterface)]) and implemented by this shared instance
- as a consequence, instance implementation pattern will always be sicShared
- will return the first of the registered TServiceFactoryServer created on success (i.e. the one corresponding to the first item of the aInterfaces array), or nil if registration failed (e.g. if any of the supplied interfaces is not implemented by the given class)
- you can use the returned TServiceFactoryServer instance to set the expected security parameters associated with this interface
- the same implementation class can be used to handle several interfaces (just as Delphi allows to do natively)
function ServiceRegister(aClient: TSQLRest; const aInterfaces: array of PTypeInfo; aInstanceCreation: TServiceInstanceImplementation=sicSingle; const aContractExpected: RawUTF8=''): boolean; overload;

Register a remote Service via its interface
- this overloaded method will register a remote Service, accessed via the supplied TSQLRest(ClientURI) instance: it can be available in the main TSQLRestServer.Services property, but execution will take place on a remote server - may be used e.g. for dedicated hosting of services (in a DMZ for instance)
- this methods expects a list of interfaces to be registered to the client (e.g. TypeInfo(IMyInterface))
- instance implementation pattern will be set by the appropriate parameter
- will return true on success, false if registration failed (e.g. if any of the supplied interfaces is not correct or is not available on the server)
- that is, server side will be called to check for the availability of each interface
- you can specify an optional custom contract for the first interface

function ServicesPublishedInterfaces: RawUTF8;

Compute a JSON description of all available services, and its public URI
- the JSON object matches the TServicesPublishedInterfaces record type
- used by TSQLRestClientURI.ServicePublishOwnInterfaces to register all the services supported by the client itself
- warning: the public URI should have been set via SetPublicURI()

function SessionGetUser(aSessionID: Cardinal): TSQLAuthUser;

Returns a copy of the user associated to a session ID
- returns nil if the session does not exist (e.g. if authentication is disabled)
- caller MUST release the TSQLAuthUser instance returned (if not nil)
- this method IS thread-safe, and calls internaly Sessions.Lock (the returned TSQLAuthUser is a private copy from Sessions[].User instance, in order to be really thread-safe)
- the returned TSQLAuthUser instance will have GroupRights=nil but will have ID, LogonName, DisplayName, PasswordHashHexa and Data fields available

function SessionsAsJson: RawJSON;

Retrieve all current session information as a JSON array

function SleepOrShutdown(MS: integer): boolean;

Wait for the specified number of milliseconds
- if Shutdown is called in-between, returns true
- if the thread waited the supplied time, returns false

function StaticDataAdd(aStaticData: TSQLRestStorage): boolean;

Register an external static storage for a given table
- will be added to StaticDataServer[] internal list
- called e.g. by StaticDataCreate(), RemoteDataCreate() or StaticMongoDBRegister()
function StaticDataCreate(aClass: TSQLRecordClass; const aFileName: TFileName = ''; aBinaryFile: boolean=false; aServerClass: TSQLRestStorageInMemoryClass= nil): TSQLRestStorage;

Create an external static in-memory database for a specific class
- call it just after Create, before TSQLRestServerDB.CreateMissingTables; warning: if you don’t call this method before CreateMissingTable method is called, the table will be created as a regular table by the main database engine, and won’t be static
- can load the table content from a file if a file name is specified (could be either JSON or compressed Binary format on disk)
- you can define a particular external engine by setting a custom class - by default, it will create a TSQLRestStorageInMemory instance
- this data handles basic REST commands, since no complete SQL interpreter can be implemented by TSQLRestStorage; to provide full SQL process, you should better use a Virtual Table class, inheriting e.g. from TSQLRecordVirtualTableAutoID associated with TSQLVirtualTableJSON/Binary via a Model.VirtualTableRegister() call before TSQLRestServer.Create
- return nil on any error, or an EModelException if the class is not in the database model

function StatsAsDocVariant(Flags: TSQLRestServerAddStats=[withTables..withSessions]): variant;

Compute the statistics about this server, as a TDocVariant document
- is a wrapper around the Stats() method-based service

function StatsAsJson(Flags: TSQLRestServerAddStats=[withTables..withSessions]): RawUTF8; virtual;

Compute the statistics about this server, as JSON
- is a wrapper around the Stats() method-based service

function TableHasRows(Table: TSQLRecordClass): boolean; override;

Overridden method for direct static class call (if any)

function TableRowCount(Table: TSQLRecordClass): Int64; override;

Overridden method for direct static class call (if any)

function UnLock(Table: TSQLRecordClass; aID: TID): boolean; override;

Implement Server-Side TSQLRest unlocking
- to be called e.g. after a Retrieve() with forupdate=TRUE
- implements our custom UNLOCK REST-like verb
- locking is handled by TSQLServer.Model
- returns true on success

function UpdateBlobFields(Value: TSQLRecord): boolean; override;

Update all BLOB fields of the supplied Value
- this overridden method will execute the direct static class, if any

procedure Auth(Ctxt: TSQLRestServerURIContext);

REST service accessible from ModelRoot/Auth URI
- called by the clients for authentication and session management
- this method won’t require an authenticated client, since it is used to initiate authentication
- this global callback method is thread-safe
**procedure** Batch(Ctxt: TSQLRestServerURIContext);

---

**REST service accessible from the ModelRoot/Batch URI**
- will execute a set of RESTful commands, in a single step, with optional automatic SQL transaction generation
- this method will require an authenticated client, for safety
- expect input as JSON commands:
  ```json
  "Table": ["cmd": values, ...]
  ```
  or for multiple tables:
  ```json
  ["cmd@Table": values, ...]
  ```
  with cmd in POST/PUT with {object} as value or DELETE with ID
- returns an array of integers: ['200', '200', ...] or ['"OK"'] if all returned status codes are 200 (HTTP_SUCCESS)
- URI are either 'ModelRoot/TableName/Batch' or 'ModelRoot/Batch'

---

**procedure** CacheFlush(Ctxt: TSQLRestServerURIContext);

---

**REST service accessible from the ModelRoot/CacheFlush URI**
- it will flush the server result cache
- this method shall be called by the clients when the Server cache may be not consistent any more (e.g. after a direct write to an external database)
- this method will require an authenticated client, for safety
- GET ModelRoot/CacheFlush URI will flush the whole Server cache, for all tables
- GET ModelRoot/CacheFlush/TableName URI will flush the specified table cache
- GET ModelRoot/CacheFlush/TableName/TableID URI will flush the content of the specified record
- POST ModelRoot/CacheFlush/_callback_ URI will be called by the client to notify the server that an interface callback instance has been released
- POST ModelRoot/CacheFlush/_ping_ URI will be called by the client after every half session timeout (or at least every hour) to notify the server that the connection is still alive

---

**procedure** Stat(Ctxt: TSQLRestServerURIContext);

---

**REST service accessible from ModelRoot/Stat URI to gather detailed information**
- returns the current execution statistics of this server, as a JSON object
- this method will require an authenticated client, for safety
- by default, will return the high-level information of this server
- will return human-readable JSON layout if ModelRoot/Stat/json is used, or the corresponding XML content if ModelRoot/Stat/xml is used
- you can define withtables, withmethods, withinterfaces, withsessions or withsqlite3 additional parameters to return detailed information about method-based services, interface-based services, per session statistics, or prepared SQLite3 SQL statement timing (for a TSQLRestServerDB instance)
- defining a 'withall' parameter will retrieve all available statistics
- note that TSQLRestServer.StatLevels property will enable statistics gathering for tables, methods, interfaces, sqlite3 or sessions
- a specific findservice=ServiceName parameter will not return any statistics, but matching URIs from the server AssociatedServices list
procedure Timestamp(Ctxt: TSQLRestServerURIContext);

REST service accessible from the ModelRoot/Timestamp URI
- returns the server time stamp TTimeLog/Int64 value as UTF-8 text
- this method will not require an authenticated client
- hidden ModelRoot/Timestamp/info command will return basic execution information, less verbose (and sensitive) than Stat(), calling virtual InternalInfo() protected method

procedure AdministrationExecute(const DatabaseName, SQL: RawUTF8; var result: TServiceCustomAnswer); override;

Used e.g. by IAdministratedDaemon to implement "pseudo-SQL" commands

procedure AnswerToMessage(var Msg: TWMCopyData); message WM_COPYDATA;

Implement a message-based server response
- this method is called automatically if ExportServerMessage() method was initialized
- you can also call this method from the WM_COPYDATA message handler of your main form, and use the TSQLRestClientURIMessage class to access the server instance from your clients
- it will answer to the Client with another WM_COPYDATA message
- message oriented architecture doesn't need any thread, but will use the main thread of your application

procedure AuthenticationRegister(const aMethods: array of TSQLRestServerAuthenticationClass); overload;

Call this method to add several authentication methods to the server
- if TSQLRestServer.Create() constructor is called with aHandleUserAuthentication set to TRUE, it will register the two following classes:

AuthenticationRegister([TSQLRestServerAuthenticationDefault, TSQLRestServerAuthenticationSSPI]);

procedure Authentication_unregister(const aMethods: array of TSQLRestServerAuthenticationClass); overload;

Call this method to remove several authentication methods to the server

procedure Authentication_unregister(aMethod: TSQLRestServerAuthenticationClass); overload;

Call this method to remove an authentication method to the server

procedure Authentication_unregisterAll;

Call this method to remove all authentication methods to the server

procedure BeginCurrentThread(Sender: TThread); override;

You can call this method in TThread.Execute to ensure that the thread will be taken into account during process
- caller must specify the TThread instance running
- used e.g. for optExecInMainThread option in TServiceMethodExecute
- this default implementation will call the methods of all its internal TSQLRestStorage instances
- this method shall be called from the thread just initiated: e.g. if you call it from the main thread, it may fail to prepare resources
procedure Commit(SessionID: cardinal; RaiseException: boolean); override;

End a transaction
- implements REST END collection
- write all pending TSQLVirtualTableJSON data to the disk

procedure CreateMissingTables(user_version: cardinal=0; options: TSQLInitializeTableOptions=[]); virtual;

Missing tables are created if they don't exist yet for every TSQLRecord class of the Database Model
- you must call explicitly this before having called StaticDataCreate()
- all table description (even Unique feature) is retrieved from the Model
- this method should also create additional fields, if the TSQLRecord definition has been modified; only field adding is mandatory, field renaming or field deleting are not allowed in the FrameWork (in such cases, you must create a new TSQLRecord type)
- this virtual method do nothing by default - overridden versions should implement it as expected by the underlying storage engine (e.g. SQLite3 or TSQLRestServerFullInMemory)
- you can tune some options transmitted to the TSQLRecord.InitializeTable virtual methods, e.g. to avoid the automatic create of indexes

procedure EndCurrentThread(Sender: TThread); override;

You can call this method just before a thread is finished to ensure e.g. that the associated external DB connection will be released
- this default implementation will call the methods of all its internal TSQLRestStorage instances, allowing e.g. TSQLRestStorageExternal instances to clean their thread-specific connections
- this method shall be called from the thread about to be terminated: e.g. if you call it from the main thread, it may fail to release resources
- it is set e.g. by TSQLite3HttpServer to be called from HTTP threads, or by TSQLRestServerNamedPipeResponse for named-pipe server cleaning

procedure FlushInternalDBCache; virtual;

Call this method when the internal DB content is known to be invalid
- by default, all REST/CRUD requests and direct SQL statements are scanned and identified as potentially able to change the internal SQL/JSON cache used at SQLite3 database level; but some virtual tables (e.g. TSQLRestStorageExternal classes defined in mORMotDB) could flush the database content without proper notification
- this default implementation will just do nothing, but mORMotSQLite3 unit will call TSQLDataBase.CacheFlush method

procedure InitializeTables(Options: TSQLInitializeTableOptions);

Run the TSQLRecord.InitializeTable methods for all void tables of the model
- can be used instead of CreateMissingTables e.g. for MongoDB storage
- you can specify the creation options, e.g. INITIALIZETABLE_NOINDEX

procedure ServiceMethodRegister(aMethodName: RawUTF8; const aEvent: TSQLRestServerCallBack; aByPassAuthentication: boolean=false);

Direct registration of a method for a given low-level event handler

procedure ServiceMethodRegisterPublishedMethods(const aPrefix: RawUTF8; aInstance: TObject);

Add all published methods of a given object instance to the method-based list of services
- all those published method signature should match TSQLRestServerCallBack
procedure SessionsLoadFromFile(const aFileName: TFileName;
andDeleteExistingFileAfterRead: boolean);

Re-create all in-memory sessions from a compressed binary file
- typical use is after a server restart, with the file supplied to the Shutdown() method: it could be used e.g. for a short maintenance server shutdown, without loosing the current logged user sessions
- WARNING: this method will restore authentication sessions for the ORM, but not any complex state information used by interface-based services, like sicClientDriven class instances - DO NOT use this feature with SOA
- this method IS thread-safe, and call internally Sessions.Lock

procedure SessionsSaveToFile(const aFileName: TFileName);

Persist all in-memory sessions into a compressed binary file
- you should not call this method directly, but rather use Shutdown() with a StateFileName parameter - to be used e.g. for a short maintenance server shutdown, without loosing the current logged user sessions
- this method IS thread-safe, and call internally Sessions.Lock

procedure SetPublicURI(const Address,Port: RawUTF8);

The HTTP server should call this method so that ServicesPublishedInterfaces registration will be able to work

procedure Shutdown(const aStateFileName: TFileName=''); virtual;

You can call this method to prepare the server for shutting down
- it will reject any incoming request from now on, and will wait until all pending requests are finished, for proper server termination
- you could optionally save the current server state (e.g. user sessions) into a file, ready to be retrieved later on using SessionsLoadFromFile - note that this will work only for ORM sessions, NOT complex SOA state
- this method is called by Destroy itself
procedure TrackChanges(const aTable: array of TSQLRecordClass; aTableHistory: TSQLRecordHistoryClass=nil; aMaxHistoryRowBeforeBlob: integer=1000; aMaxHistoryRowPerRecord: integer=10; aMaxUncompressedBlobSize: integer=64*1024); virtual;

Initialize change tracking for the given tables
- by default, it will use the TSQLRecordHistory table to store the changes - you can specify a dedicated class as aTableHistory parameter
- if aTableHistory is not already part of the TSQLModel, it will be added
- note that this setting should be consistent in time: if you disable tracking for a while, or did not enable tracking before adding a record, then the content history won't be consistent (or disabled) for this record
- at every change, aTableHistory.SentDataJSON records will be added, up to aMaxHistoryRowBeforeBlob items - then aTableHistory.History will store a compressed version of all previous changes
- aMaxHistoryRowBeforeBlob is the maximum number of JSON rows per Table before compression into BLOB is triggered
- aMaxHistoryRowPerRecord is the maximum number of JSON rows per record, above which the versions will be compressed as BLOB
- aMaxUncompressedBlobSize is the maximum BLOB size per record
- you can specify aMaxHistoryRowBeforeBlob=0 to disable change tracking
- you should call this method after the CreateMissingTables call
- note that change tracking may slow down the writing process, and may increase storage space a lot (even if BLOB maximum size can be set), so should be defined only when necessary

procedure TrackChangesFlush(aTableHistory: TSQLRecordHistoryClass); virtual;

Force compression of all aTableHistory.SentDataJSON into History BLOB
- by default, this will take place in InternalUpdateEvent() when aMaxHistoryRowBeforeBlob - as set by TrackChanges() method - is reached
- you can manually call this method to force History BLOB update, e.g. when the server is in Idle state, and ready for process

procedure URI(var Call: TSQLRestURIParams); virtual;
Implement a generic local, piped or HTTP/1.1 provider
- this is the main entry point of the server, from the client side
- default implementation calls protected methods EngineList() Retrieve() Add() Update() Delete() UnLock() EngineExecute() above, which must be overridden by the TSQLRestServer child
- for 'GET ModelRoot/TableName', url parameters can be either "select" and "where" (to specify a SQL Query, from the SQLFromSelectWhere function), either "sort", "dir", "startIndex", "results", as expected by the YUI DataSource Request Syntax for data pagination - see http://developer.yahoo.com/yui/datatable/#data
- execution of this method could be monitored via OnBeforeURI and OnAfterURI event handlers

Used for DI-2.1.1.2.4 (page 2545).

property AssociatedServices: TServicesPublishedInterfacesList read fAssociatedServices;

A list of the services associated by all clients of this server instance
- when a client connects to this server, it will publish its own services (when checking its interface contract), so that they may be identified
property AuthenticationSchemes: TSQLRestServerAuthenticationDynArray read fSessionAuthentication;

Read-only access to the list of registered server-side authentication methods, used for session creation
- note that the exact number or registered services in this list is stored in the AuthenticationSchemesCount property

property AuthenticationSchemesCount: integer read GetAuthenticationSchemesCount;

How many authentication methods are registered in AuthenticationSchemes

property BypassORMAuthentication: TSQLURIMethods read fBypassORMAuthentication write fBypassORMAuthentication;

Allow to by-pass Authentication for a given set of HTTP verbs
- by default, RESTful access to the ORM will follow HandleAuthentication setting: but you could define some HTTP verb to this property, which will by-pass the authentication - may be used e.g. for public GET of the content by an AJAX client

property CreateMissingTablesOptions: TSQLInitializeTableOptions read fCreateMissingTablesOptions;

The options specified to TSQLRestServer.CreateMissingTables
- as expected by TSQLRecord.InitializeTable methods

property HandleAuthentication: boolean read fHandleAuthentication;

Set to true if the server will handle per-user authentication and access right management
- i.e. if the associated TSQLModel contains TSQLAuthUser and TSQLAuthGroup tables (set by constructor)

property JWTForUnauthenticatedRequest: TJWTAbstract read fJWTForUnauthenticatedRequest write fJWTForUnauthenticatedRequest;

Define if unsecure connections (i.e. not in-process or encrypted WebSockets) with no session can be authenticated via JWT
- once set, this instance will be owned by the TSQLRestServer
- by definition, such JWT authentication won't identify any mORMot user nor session (it just has to be valid), so only sicSingle, sicShared or sicPerThread interface-based services execution are possible
- typical usage is for a public API, in conjunction with ServiceDefine(...).ResultAsJSONObjectWithoutResult := true on the server side and TSQLRestClientURI.ServiceDefineSharedAPI() method for the client
- see also JWTForUnauthenticatedRequestWhiteIP() for additional security
**property** NoAJAXJSON: boolean read GetNoAJAXJSON write SetNoAJAXJSON;

*Set this property to true to transmit the JSON data in a "not expanded" format*

- not directly compatible with Javascript object list decode: not to be used in AJAX environment (like in TSQLite3HttpServer)
- but transmitted JSON data is much smaller if set it's set to FALSE, and if you use a Delphi Client, parsing will be also faster and memory usage will be lower
- By default, the NoAJAXJSON property is set to TRUE in TSQLRestServer.ExportServerNamedPipe: if you use named pipes for communication, you probably won't use javascript because browser communicates via HTTP!
- But otherwise, NoAJAXJSON property is set to FALSE. You could force its value to TRUE and you'd save some bandwidth if you don't use javascript: even the parsing of the JSON Content will be faster with Delphi client if JSON content is not expanded
- the "expanded" or standard/AJAX layout allows you to create pure JavaScript objects from the JSON content, because the field name / JavaScript object property name is supplied for every value
- the "not expanded" layout, NoAJAXJSON property is set to TRUE, reflects exactly the layout of the SQL request - first line contains the field names, then all next lines are the field content
- is in fact stored in rsoNoAJAXJSON item in Options property

**property** Options: TSQLRestServerOptions read fOptions write fOptions;

*Allow to customize how TSQLRestServer.URI process the requests*

- e.g. if HTTP_SUCCESS with no body should be translated into HTTP_NOCONTENT

**property** RecordVersionDeleteIgnore: boolean read fRecordVersionDeleteIgnore write fRecordVersionDeleteIgnore;

*You can force this property to TRUE so that any Delete() will not write to the TSQLRecordTableDelete table for TRecordVersion tables*

- to be used when applying a TSQLRestBatch instance as returned by RecordVersionSynchronizeToBatch()

**property** RootRedirectGet: RawUTF8 read fRootRedirectGet write fRootRedirectGet;

*The URI to redirect any plain GET on root URI, without any method*

- could be used to ease access from web browsers URI

**property** ServiceMethodStat[const aMethod: RawUTF8]: TSynMonitorInputOutput read GetServiceMethodStat;

*Retrieve detailed statistics about a method-based service use*

- will return a reference to the actual alive item: caller should not free the returned instance

**property** SessionClass: TAuthSessionClass read fSessionClass write fSessionClass;

*The class inheriting from TAuthSession to handle in-memory sessions*

- since all sessions data remain in memory, ensure they are not taking too much resource (memory or process time)

**property** Sessions: TSynObjectListLocked read fSessions;

*Read-only access to the internal list of sessions*

- ensure you protect its access calling Sessions.Lock/Sessions.Unlock

**property** SQLAuthGroupClass: TSQLAuthGroupClass read fSQLAuthGroupClass;

*The class inheriting from TSQLAuthGroup, as defined in the model*

- during authentication, this class will be used for every TSQLAuthGroup table access
property SQLAuthUserClass: TSQLAuthUserClass read fSQLAuthUserClass;

The class inheriting from TSQLAuthUser, as defined in the model
- during authentication, this class will be used for every TSQLAuthUser table access
- see also the OnAuthenticationUserRetrieve optional event handler

property SQLRecordVersionDeleteTable: TSQLRecordTableDeletedClass read fSQLRecordVersionDeleteTable;

The class inheriting from TSQLRecordTableDeleted, as defined in the model
- during authentication, this class will be used for storing a trace of every deletion of table rows containing a TRecordVersion published field

property StaticDataServer[aClass: TSQLRecordClass]: TSQLRest read GetStaticDataServer;

Retrieve the TSQLRestStorage instance used to store and manage a specified TSQLRecordClass in memory
- has been associated by the StaticDataCreate method

property StaticTable[aClass: TSQLRecordClass]: TSQLRest read GetStaticTable;

Fast get the associated static server or virtual table, if any
- same as a dual call to StaticDataServer[aClass] + StaticVirtualTable[aClass]

property StaticVirtualTable[aClass: TSQLRecordClass]: TSQLRest read GetVirtualTable;

Retrieve a running TSQLRestStorage virtual table
- associated e.g. to a 'JSON' or 'Binary' virtual table module, or may return a TSQLRestStorageExternal instance (as defined in mORMotDB)
- this property will return nil if there is no Virtual Table associated or if the corresponding module is not a TSQLVirtualTable (e.g. "pure" static tables registered by StaticDataCreate will be accessible only via StaticDataServer[], not via StaticVirtualTable[])  
- has been associated by the TSQLModel.VIRTUAL TABLE Register method or the VirtualTableExternalRegister() global function

property StaticVirtualTableDirect: boolean read fVirtualTableDirect write fVirtualTableDirect;

This property can be left to its TRUE default value, to handle any TSQLVirtualTableJSON static tables (module JSON or BINARY) with direct calls to the storage instance
- is set to TRUE by default to enable faster Direct mode
- in Direct mode, GET/POST/PUT/DELETE of individual records (or BLOB fields) from URI() will call directly the corresponding TSQLRestStorage instance, for better speed for most used RESTful operations; but complex SQL requests (e.g. joined SELECT) will rely on the main SQL engine
- if set to false, will use the main SQLite3 engine for all statements (should not to be used normally, because it will add unnecessary overhead)

property StatLevels: TSQLRestServerMonitorLevels read fStatLevels write fStatLevels;

Which level of detailed information is gathered
- by default, contains SERVER DEFAULT MONITOR LEVELS, i.e. [mlTables, mlMethods, mlInterfaces, mlSQLite3]
- you can add mlSessions to maintain per-session statistics: this will lead into a slightly higher memory consumption, for each session
property Stats: TSQLRestServerMonitor read fStats;

- **Read-only access to the high-level Server statistics**
- see ServiceMethodStat[] for information about method-based services, or TServiceFactoryServer.Stats / Stat[] for interface-based services
- statistics are available remotely as JSON from the Stat() method

property StatUsage: TSynMonitorUsage read fStatUsage write SetStatUsage;

- **Could be set to track statistic from Stats information**
- it may be e.g. a TSynMonitorUsageRest instance for REST storage

TSQLRestStorage = class(TSQLRest)

- **REST class with direct access to an external database engine**
- you can set an alternate per-table database engine by using this class
- this abstract class is to be overridden with a proper implementation (e.g. TSQLRestStorageInMemory in this unit, or TSQLRestStorageExternal from mORMotDB unit, or TSQLRestStorageMongoDB from mORMotMongoDB unit)

constructor Create(aClass: TSQLRecordClass; aServer: TSQLRestServer); reintroduce; virtual;

- Initialize the abstract storage data

destructor Destroy; override;

- Finalize the storage instance

function CreateSQLMultiIndex(Table: TSQLRecordClass; const FieldNames: array of RawUTF8; Unique: boolean; IndexName: RawUTF8=''): boolean; virtual;

- Create one index for all specific FieldNames at once
- do nothing method: will return FALSE (aka error)

function RecordCanBeUpdated(Table: TSQLRecordClass; ID: TID; Action: TSQLEvent; ErrorMsg: PRawUTF8 = nil): boolean; override;

- Overridden method calling the owner (if any) to guess if this record can be updated or deleted

function SearchField(const FieldName: RawUTF8; FieldValue: Int64; out ResultID: TIDDynArray): boolean; overload; virtual;

- Search for a numerical field value
- return true on success (i.e. if some values have been added to ResultID)
- store the results into the ResultID dynamic array
- faster than OneFieldValues method, which creates a temporary JSON content
- this default implementation will call the overloaded SearchField() value after conversion of the FieldValue into RawUTF8

function SearchField(const FieldName, FieldValue: RawUTF8; out ResultID: TIDDynArray): boolean; overload; virtual; abstract;

- Search for a field value, according to its SQL content representation
- return true on success (i.e. if some values have been added to ResultID)
- store the results into the ResultID dynamic array
- faster than OneFieldValues method, which creates a temporary JSON content
function ServiceContainer: TServiceContainer; override;

Access or initialize the internal IoC resolver
- this overridden method will return always nil, since IoC only makes sense at TSQLRestClient and TSQLRestServer level

function UnLock(Table: TSQLRecordClass; aID: TID): boolean; override;

Implement TSQLRest unlocking (UNLOCK verb)
- to be called e.g. after a Retrieve() with forupdate=TRUE
- locking is handled at (Owner.)Model level
- returns true on success

procedure BeginCurrentThread(Sender: TThread); override;

You can call this method in TThread.Execute to ensure that the thread will be taken into account during process
- this overridden method will do nothing (should have been already made at TSQLRestServer caller level)
- children classes may inherit from this method to notify e.g. a third party process (like proper OLE initialization)

procedure EndCurrentThread(Sender: TThread); override;

You can call this method just before a thread is finished to ensure e.g. that the associated external DB connection will be released
- this overridden method will do nothing (should have been already made at TSQLRestServer caller level)
- children classes may inherit from this method to notify e.g. a third party process (like proper OLE initialization)

procedure StorageLock(WillModifyContent: boolean; const msg: RawUTF8); virtual;

Should be called before any access to the storage content
- and protected with a try ... finally StorageUnLock; end section

procedure StorageUnLock; virtual;

Should be called after any StorageLock-protected access to the content
- e.g. protected with a try ... finally StorageUnLock; end section

property Modified: boolean read fModified write fModified;

Read only access to a boolean value set to true if table data was modified

property Owner: TSQLRestServer read fOwner;

Read only access to the TSQLRestServer using this storage engine

property StorageLockLogTrace: boolean read fStorageLockLogTrace write fStorageLockLogTrace;

Enable low-level trace of StorageLock/StorageUnlock methods
- may be used to resolve low-level race conditions

property StoredClass: TSQLRecordClass read fStoredClass;

Read only access to the class defining the record type stored in this REST storage

property StoredClassName: RawUTF8 read GetStoredClassName;

Name of the class defining the record type stored in this REST storage
property StoredClassProps: TSQLModelRecordProperties read fStoredClassProps; 
Read only access to the ORM properties of the associated record type 
- may be nil if this instance is not associated with a TSQLModel

property StoredClassRecordProps: TSQLRecordProperties read fStoredClassRecordProps; 
Read only access to the RTTI properties of the associated record type

TSQLRestStorageRecordBased = class(TSQLRestStorage)
Abstract REST storage exposing some internal TSQLRecord-based methods

function AddOne(Rec: TSQLRecord; ForceID: boolean; const SentData: RawUTF8): TID; virtual; abstract;
Manual Add of a TSQLRecord
- returns the ID created on success 
- returns -1 on failure (not UNIQUE field value e.g.) 
- on success, the Rec instance is added to the Values[] list: caller doesn't need to Free it

function GetOne(aID: TID): TSQLRecord; virtual; abstract;
Manual Retrieval of a TSQLRecord field values 
- an instance of the associated static class is created 
- and all its properties are filled from the Items[] values 
- caller can modify these properties, then use UpdateOne() if the changes have to be stored inside the Items[] list 
- caller must always free the returned instance 
- returns NIL if any error occurred, e.g. if the supplied aID was incorrect 
- method available since a TSQLRestStorage instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

function UpdateOne(ID: TID; const Values: TSQLVarDynArray): boolean; overload; virtual;
Manual Update of a TSQLRecord field values from an array of TSQLVar 
- will update all properties, including BLOB fields and such 
- returns TRUE on success, FALSE on any error (e.g. invalid Rec.ID) 
- method available since a TSQLRestStorage instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer 
- this default implementation will create a temporary TSQLRecord instance with the supplied Values[], and will call overloaded UpdateOne() method

function UpdateOne(Rec: TSQLRecord; const SentData: RawUTF8): boolean; overload; virtual; abstract;
Manual Update of a TSQLRecord field values 
- Rec.ID specifies which record is to be updated 
- will update all properties, including BLOB fields and such 
- returns TRUE on success, FALSE on any error (e.g. invalid Rec.ID) 
- method available since a TSQLRestStorage instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer
TSQLRestStorageInMemoryUnique = class(TObject)

  Class able to handle a O(1) hashed-based search of a property
  - used e.g. to hash TSQLRestStorageInMemory field values

  constructor Create(aOwner: TSQLRestStorageInMemory; aField: TSQLPropInfo);

  Match TEventDynArrayHashOne initialize a hash for a record array field
  - aField maps the "stored AS_UNIQUE" published property

function AddedAfterFind(Rec: TSQLRecord): boolean;
  Called by TSQLRestStorageInMemory.AddOne after a precious Find()

function Find(Rec: TSQLRecord): integer;
  Fast search using O(1) internal hash table
  - returns -1 if not found or not indexed (self=nil)

property CaseInsensitive: boolean read fCaseInsensitive;
  If the string comparison shall be case-insensitive

property Hasher: TDynArrayHasher read fHasher;
  Access to the internal hash table

property PropInfo: TSQLPropInfo read fPropInfo;
  The corresponding field RTTI

TSQLRestStorageInMemory = class(TSQLRestStorageRecordBased)

  REST storage with direct access to a TObjectList memory-stored table
  - store the associated TSQLRecord values in memory
  - handle one TSQLRecord per TSQLRestStorageInMemory instance
  - must be registered individually in a TSQLRestServer to access data from a common client, by
    using the TSQLRestServer.StaticDataCreate method: it allows an unique access for both SQLite3
    and Static databases
  - handle basic REST commands, no SQL interpreter is implemented: only valid SQL command is
    "SELECT Field1,Field2 FROM Table WHERE ID=120;", i.e a one Table SELECT with one optional
    "WHERE fieldname = value" statement; if used within a TSQLVirtualTableJSON, you'll be able to
    handle any kind of SQL statement (even joined SELECT or such) with this memory-stored database
    - our TSQLRestStorage database engine is very optimized and is a lot faster than SQLite3 for such
      queries - but its values remain in RAM, therefore it is not meant to deal with more than 100,000
      rows
    - data can be stored and retrieved from a file (JSON format is used by default, if BinaryFile
      parameter is left to false; a proprietary compressed binary format can be used instead) if a file
      name is supplied at creating the TSQLRestStorageInMemory instance

  constructor Create(aClass: TSQLRecordClass; aServer: TSQLRestServer; const
  aFileName: TFileName = ''; aBinaryFile: boolean=false); reintroduce; virtual;

  Initialize the table storage data, reading it from a file if necessary
  - data encoding on file is UTF-8 JSON format by default, or should be some binary format if
    aBinaryFile is set to true
**Synops e mORMot Framework**  
**Software Architecture Design 1.18**  
**Date: September 16, 2020**

```cpp
// Destructor
Destroy; override;
Free used memory
- especially release all fValue[] instances

// Function
AddOne(Rec: TSQLRecord; ForceID: boolean; const SentData: RawUTF8): TID; override;

Low-level Add of a TSQLRecord instance
- returns the ID created on success
- returns -1 on failure (not UNIQUE field value e.g.)
- on success, the Rec instance is added to the Values[] list: caller doesn't need to Free it, since it will be owned by the storage
- in practice, SentData is used only for OnUpdateEvent/OnBlobUpdateEvent and the history feature
- warning: this method should be protected via StorageLock/StorageUnlock

// Function
DeleteOne(aIndex: integer): boolean; virtual;

Direct deletion of a TSQLRecord, from its index in Values[]
- warning: this method should be protected via StorageLock/StorageUnlock

// Function
EngineDelete(TableModelIndex: integer; ID: TID): boolean; override;

Overridden method for direct in-memory database engine call
- made public since a TSQLRestStorage instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

// Function
EngineUpdateField(TableModelIndex: integer; const SetFieldName, SetValue, WhereFieldName, WhereValue: RawUTF8): boolean; override;

Overridden method for direct in-memory database engine call
- made public since a TSQLRestStorage instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

// Function
EngineUpdateFieldIncrement(TableModelIndex: integer; ID: TID; const FieldName: RawUTF8; Increment: Int64): boolean; override;

Overridden method for direct in-memory database engine call
- made public since a TSQLRestStorage instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

// Function
FindMax(WhereField: integer; out max: Int64): boolean;

Search the maximum value of a given column
- will only handle integer/Int64 kind of column

// Function
FindWhereEqual(WhereField: integer; const WhereValue: RawUTF8; OnFind: TFindWhereEqualEvent; Dest: pointer; FoundLimit,FoundOffset: PtrInt; CaseInsensitive: boolean=true): PtrInt; overload;

Optimized search of WhereValue in WhereField (0=RowID,1..=RTTI)
- will use fast O(1) hash for fUnique[] fields
- will use SYSTEMNOCASE case-insensitive search for text values, unless CaseInsensitive is set to FALSE
- warning: this method should be protected via StorageLock/StorageUnlock
```
function FindWhereEqual(const WhereFieldName, WhereValue: RawUTF8; OnFind: TFindWhereEqualEvent; Dest: pointer; FoundLimit, FoundOffset: integer; CaseInsensitive: boolean=true): PtrInt; overload;

*Optimized search of WhereValue in a field, specified by name*
- will use fast O(1) hash for fUnique[] fields
- will use SYSTEMNOCASE case-insensitive search for text values, unless CaseInsensitive is set to FALSE
- warning: this method should be protected via StorageLock/StorageUnlock

function GetOne(aID: TID): TSQLRecord; override;

*Manual Retrieval of a TSQLRecord field values*
- an instance of the associated static class is created, and filled with the actual properties values
- and all its properties are filled from the Items[] values
- caller can modify these properties, then use UpdateOne() if the changes have to be stored inside the Items[] list
- caller must always free the returned instance
- returns NIL if any error occurred, e.g. if the supplied aID was incorrect
- method available since a TSQLRestStorage instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

function IDToIndex(ID: TID): PtrInt;

*Retrieve the index in Items[] of a particular ID*
- return -1 if this ID was not found
- use internally fast O(1) hashed search algorithm
- warning: this method should be protected via StorageLock/StorageUnlock

function LoadFromBinary(Stream: TStream): boolean; overload;

*Load the values from binary file/stream*
- the binary format is a custom compressed format (using our SynLZ fast compression algorithm), with variable-length record storage
- the binary content is first checked for consistency, before loading
- warning: the field layout should be the same at SaveToBinary call; for instance, it won’t be able to read a file content with a renamed or modified field type
- will return false if the binary content is invalid

function LoadFromBinary(const Buffer: RawByteString): boolean; overload;

*Load the values from binary data*
- uses the same compressed format as the overloaded stream/file method
- will return false if the binary content is invalid

function MemberExists(Table: TSQLRecordClass; ID: TID): boolean; override;

*Overridden method for direct in-memory database engine call*

function RetrieveBlobFields(Value: TSQLRecord): boolean; override;

*Overridden method for direct in-memory database engine call*
function SaveToBinary(Stream: TStream): integer; overload;
  Save the values into a binary file/stream
  - the binary format is a custom compressed format (using our SynLZ fast compression
    algorithm), with variable-length record storage: e.g. a 27 KB Dali1.json content is stored into a 6
    KB Dali2.data file (this data has a text redundant field content in its FirstName field); 502 KB
    People.json content is stored into a 92 KB People.data file
  - returns the number of bytes written into Stream

function SaveToBinary: RawByteString; overload;
  Save the values into a binary buffer
  - uses the same compressed format as the overloaded stream/file method

function SaveToJSON(Expand: Boolean): RawUTF8; overload;
  Save the values into JSON data

function SearchCopy(const FieldName, FieldValue: RawUTF8): pointer;
  Search for a field value, according to its SQL content representation
  - return a copy of the found TSQLRecord on success, nil if no match
  - you should use SearchCopy() instead of SearchInstance(), unless you are sure that the internal
    TSQLRecord list won't change

function SearchCount(const FieldName, FieldValue: RawUTF8): integer;
  Search and count for a field value, according to its SQL content representation
  - return the number of found entries on success, 0 if it was not found

function SearchEvent(const FieldName, FieldValue: RawUTF8; OnFind:
  TFindWhereEqualEvent; Dest: pointer; FoundLimit,FoundOffset:_PTRInt): integer;
  Search for a field value, according to its SQL content representation
  - call the supplied OnFind event on match
  - returns the number of found entries
  - is just a wrapper around FindWhereEqual() with StorageLock protection

function SearchField(const FieldName, FieldValue: RawUTF8; out ResultID:
  TIDDynArray): boolean; override;
  Search for a field value, according to its SQL content representation
  - return true on success (i.e. if some values have been added to ResultID)
  - store the results into the ResultID dynamic array
  - faster than OneFieldValues method, which creates a temporary JSON content

function SearchIndex(const FieldName, FieldValue: RawUTF8): integer;
  Search for a field value, according to its SQL content representation
  - return the found TSQLRecord index on success, -1 if none did match
  - warning: it returns a reference to the current index of the unlocked internal list, so you should
    NOT use without StorageLock/StorageUnlock

function SearchInstance(const FieldName, FieldValue: RawUTF8): pointer;
  Search for a field value, according to its SQL content representation
  - return the found TSQLRecord on success, nil if none did match
  - warning: it returns a reference to one item of the unlocked internal list, so you should NOT use
    this on a read/write table, but rather use the slightly slower but safer SearchCopy() method or
    make explicit
    StorageLock ... try ... SearchInstance ... finally StorageUnlock end
function TableHasRows(Table: TSQLRecordClass): boolean; override;
Overridden method for direct in-memory database engine call

function TableRowCount(Table: TSQLRecordClass): Int64; override;
Overridden method for direct in-memory database engine call

function UpdateBlobFields(Value: TSQLRecord): boolean; override;
Overridden method for direct in-memory database engine call

function UpdateOne(ID: TID; const Values: TSQLVarDynArray): boolean; override;
Manual Update of a TSQLRecord field values from a TSQLVar array
- will update all properties, including BLOB fields and such
- returns TRUE on success, FALSE on any error (e.g. invalid Rec.ID)
- method available since a TSQLRestStorage instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

function UpdateOne(Rec: TSQLRecord; const SentData: RawUTF8): boolean; override;
Manual Update of a TSQLRecord field values
- Rec.ID specifies which record is to be updated
- will update all properties, including BLOB fields and such
- returns TRUE on success, FALSE on any error (e.g. invalid Rec.ID)
- method available since a TSQLRestStorage instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

class procedure DoAddToListEvent(aDest: pointer; aRec: TSQLRecord; aIndex: integer);
Low-level TFindWhereEqualEvent callback calling TSynList(aDest).Add(aRec)

class procedure DoCopyEvent(aDest: pointer; aRec: TSQLRecord; aIndex: integer);
Low-level TFindWhereEqualEvent callback setting PPointer(aDest)^ := aRec.CreateCopy

class procedure DoIndexEvent(aDest: pointer; aRec: TSQLRecord; aIndex: integer);
Low-level TFindWhereEqualEvent callback setting PInteger(aDest)^ := aIndex

class procedure DoInstanceEvent(aDest: pointer; aRec: TSQLRecord; aIndex: integer);
Low-level TFindWhereEqualEvent callback setting PPointer(aDest)^ := aRec

class procedure DoNothingEvent(aDest: pointer; aRec: TSQLRecord; aIndex: integer);
Low-level TFindWhereEqualEvent callback doing nothing

procedure DropValues(andUpdateFile: boolean=true);
Clear all the values of this table
- will reset the associated database file, if any

procedure ForEach(WillModifyContent: boolean; OnEachProcess: TFindWhereEqualEvent; Dest: pointer);
Execute a method on every TSQLRecord item
- the loop execution will be protected via StorageLock/StorageUnlock

procedure GetAllIDs(out ID: TIDDynArray);
Retrieve all IDs stored at once
- will make a thread-safe copy, for unlocked use
procedure LoadFromJSON(const aJSON: RawUTF8); overload;
    Load the values from JSON data
    - a temporary copy of aJSON is made to ensure it won't be modified in-place
    - consider using the overloaded PUTF8Char/len method if you don't need this copy

procedure LoadFromJSON(JSONBuffer: PUTF8Char; JSONBufferLen: integer); overload;
    Load the values from JSON data

procedure LoadFromResource(ResourceName: string=''; Instance: THandle=0);
    Load the values from binary resource
    - the resource name is expected to be the TSQLRecord class name, with a resource type of 10
    - uses the same compressed format as the overloaded stream/file method
    - you can specify a library (dll) resource instance handle, if needed

procedure ReloadFromFile;
    Will reload all content from the current disk file
    - any not saved modification will be lost (e.g. if Updatefile has not been called since)

procedure SaveToJSON(Stream: TStream; Expand: Boolean); overload;
    Save the values into JSON data

procedure UpdateFile;
    If file was modified, the file is updated on disk
    - this method is called automatically when the TSQLRestStorage instance is destroyed: should
    - do nothing if the table content was not modified
    - will write JSON content by default, or binary content if BinaryFile property was set to true

property BinaryFile: boolean read fBinaryFile write SetBinaryFile;
    If set to true, file content on disk will expect binary format
    - default format on disk is JSON but can be overridden at constructor call
    - binary format should be more efficient in term of speed and disk usage, but can be proprietary
    - if you change manually the file format from this property, the storage will be marked as
      "modified" so that UpdateFile will save the content

property CommitShouldNotUpdateFile: boolean read fCommitShouldNotUpdateFile write fCommitShouldNotUpdateFile;
    Set this property to TRUE if you want the COMMIT statement not to update the associated
    TSQLVirtualTableJSON

property Count: integer read fCount;
    Read-only access to the number of TSQLRecord values

property ExpandedJSON: boolean read fExpandedJSON write fExpandedJSON;
    JSON writing, can set if the format should be expanded or not
    - by default, the JSON will be in the custom non-expanded format, to save disk space and time
    - you can force the JSON to be emitted as an array of objects, e.g. for better human friendliness
      (reading and modification)
Synopses mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

property FileName: TFileName read fFileName write SetFileName;
  Read only access to the file name specified by constructor
  - you can call the TSQLRestServer.StaticDataCreate method to update the file name of an
    already instanciated static table
  - if you change manually the file name from this property, the storage will be marked as
    "modified" so that UpdateFile will save the content

property ID[Index: integer]: TID read GetID;
  Read-only access to the ID of a TSQLRecord values
  - warning: this method should be protected via StorageLock/StorageUnlock

property Items[Index: integer]: TSQLRecord read GetItem;
  Read-only access to the TSQLRecord values, storing the data
  - this returns directly the item class instance stored in memory: if you change the content, it will
    affect the internal data - so for instance DO NOT change the ID values, unless you may have
    unexpected behavior
  - warning: this method should be protected via StorageLock/StorageUnlock

property Value: TSQLRecordObjArray read fValue;
  Direct access to the memory of the internal dynamic array storage
  - Items[] is preferred, since it will check the index, but is slightly slower, e.g. in a loop or after a
    IDToIndex() call
  - warning: this method should be protected via StorageLock/StorageUnlock

TSQLRestStorageInMemoryExternal = class(TSQLRestStorageInMemory)
  REST storage with direct access to a memory database, to be used as an external SQLite3 Virtual
  table
  - this is the kind of in-memory table expected by TSQLVirtualTableJSON, in order to be consistent
    with the internal DB cache

constructor Create(aClass: TSQLRecordClass; aServer: TSQLRestServer; const
  aFileName: TFileName = ''; aBinaryFile: boolean=false); override;
  Initialize the table storage data, reading it from a file if necessary
  - data encoding on file is UTF-8 JSON format by default, or should be some binary format if
    aBinaryFile is set to true

procedure StorageLock(WillModifyContent: boolean; const msg: RawUTF8); override;
  This overridden method will notify the Owner when the internal DB content is known to be
  invalid
  - by default, all REST/CRUD requests and direct SQL statements are scanned and identified as
    potentially able to change the internal SQL/JSON cache used at SQLite3 database level; but
    TSQLVirtualTableJSON virtual tables could flush the database content without proper
    notification
  - this overridden implementation will call Owner.FlushInternalDBCache
TSQLRestStorageRemote = class(TSQLRestStorage)

REST storage with redirection to another REST instance
- allows redirection of all CRUD operations for a table to another TSQLRest instance, may be a remote TSQLRestClient or a TSQLRestServer
- will be used by TSQLRestServer.RemoteDataCreate() method

constructor Create(aClass: TSQLRecordClass; aServer: TSQLRestServer; aRemoteRest: TSQLRest); reintroduce; virtual;

Initialize the table storage redirection
- you should not have to use this constructor, but rather the TSQLRestServer.RemoteDataCreate() method which will create and register one TSQLRestStorageRemote instance

property RemoteRest: TSQLRest read fRemoteRest;
The remote ORM instance used for data persistence
- may be a TSQLRestClient or a TSQLRestServer instance

TSQLRestStorageShard = class(TSQLRestStorage)

Abstract REST storage with redirection to several REST instances, implementing range ID partitioning for horizontal scaling
- such database shards will allow to scale with typical BigData storage
- this storage will add items on a server, initializing a new server when the ID reached a defined range
- it will maintain a list of previous storages, then redirect reading and updates to the server managing this ID (if possible - older shards may be deleted/ignored to release resources)
- inherited class should override InitShards/InitNewShard to customize the kind of TSQLRest instances to be used for each shard (which may be local or remote, a SQLite3 engine or an external SQL/NoSQL database)
- see inherited TSQLRestStorageShardDB as defined in mORMotSQLite3.pas

constructor Create(aClass: TSQLRecordClass; aServer: TSQLRestServer; aShardRange: TID; aOptions: TSQLRestStorageShardOptions; aMaxShardCount: integer); reintroduce; virtual;

Initialize the table storage redirection for sharding
- you should not have to use this constructor, but e.g. TSQLRestStorageShardDB.Create on a main TSQLRestServer.StaticDataAdd()
- the supplied aShardRange should be < 1000 - and once set, you should NOT change this value on an existing shard, unless process will be broken

destructor Destroy; override;

Finalize the table storage, including Shards[] instances

function ShardFromID(aID: TID; out aShardTableIndex: integer; out aShard: TSQLRest; aOccasion: TSQLOccasion=sSelect; aShardIndex: PInteger=nil): boolean; virtual;

Retrieve the ORM shard instance corresponding to an ID
- may return false if the correspondig shard is not available any more
- may return true, and a TSQLRestHookClient or a TSQLRestHookServer instance with its associated index in TSQLRest.Model.Tables[]
- warning: this method should be protected via StorageLock/StorageUnlock
function TableHasRows(Table: TSQLRecordClass): boolean; override;
    // Check if there is some data rows in a specified table

function TableRowCount(Table: TSQLRecordClass): Int64; override;
    // Get the row count of a specified table

procedure ConsolidateShards; virtual;
    // You may call this method sometimes to consolidate the sharded data
    // - may e.g. merge/compact shards, depending on scaling expectations
    // - also called by Destroy - do nothing by default

procedure RemoveShard(aShardIndex: integer); virtual;
    // Remove a shard database from the current set
    // - it will allow e.g. to delete a *.dbs file at runtime, without restarting the server
    // - this default implementation will free and nil fShard[aShardIndex], which is enough for most implementations (e.g. TSQLRestStorageShardDB)

property MaxShardCount: integer read fMaxShardCount;
    // How many shards should be maintained at most
    // - if some older shards are available on disk, they won't be loaded by InitShards, and newly added shard via InitNewShard will trigger RemoveShard if the total number of shards

property Options: TSQLRestStorageShardOptions read fOptions write fOptions;
    // Defines how this instance will handle its sharding process
    // - by default, update/delete operations or per ID retrieval will take place on all shards, whereas EngineList and EngineExecute will only run only on the latest shard (to save resources)

property ShardRange: TID read fShardRange;
    // How much IDs should store each ORM shard instance
    // - once set, you should NEVER change this value on an existing shard, otherwise the whole ID partition will fail
    // - each shard will hold [ShardIndex*ShardRange..(ShardIndex+1)*ShardRange-1] IDs

TSQLRestServerFullMemory = class(TSQLRestServer)
    // A REST server using only in-memory tables
    // - this server will use TSQLRestStorageInMemory instances to handle the data in memory, and optionally persist the data on disk as JSON or binary files
    // - so it will not handle all SQL requests, just basic CRUD commands on separated tables
    // - at least, it will compile as a TSQLRestServer without complaining for pure abstract methods; it can be used to host some services if database and ORM needs are basic (e.g. if only authentication and CRUD are needed), without the need to link the SQLite3 engine

constructor Create(aModel: TSQLModel; const aFileName: TFileName; aBinaryFile: boolean=false; aHandleUserAuthentication: boolean=false); reintroduce; overload; virtual;
    // Initialize an in-memory REST server with a database file
    // - all classes of the model will be created as TSQLRestStorageInMemory
    // - then data persistence will be initialized using aFileName, but no file will be written to disk, unless you call explicitly UpdateToFile
    // - if aFileName is left void (''), data will not be persistent
constructor Create(aModel: TSQLModel; aHandleUserAuthentication: boolean=false); overload; override;
  Initialize an in-memory REST server with no database file

constructor CreateWithOwnedAuthenticatedModel(const Tables: array of TSQLRecordClass; const aUserName, aHashedPassword: RawUTF8; aRoot: RawUTF8='root');
  Initialize an in-memory REST server with a temporary Database Model, and optional authentication by a single user
  - a Model will be created with supplied tables, and owned by the server
  - if aUserName is set, authentication will be enabled, and the supplied credentials will be used to authenticate a single user, member of the 'Supervisor' group - in this case, aHashedPassword value should match TSQLAuthUser.PasswordHashHexa expectations

destructor Destroy; override;
  Finalize the REST server
  - this overridden destructor will write any modification on file (if needed), and release all used memory

procedure Flush(Ctxt: TSQLRestServerURIContext);
  This method-base service will be accessible from ModelRoot/Flush URI, and will write any modification into file
  - method parameters signature matches TSQLRestServerCallBack type
  - do nothing if file name was not assigned
  - can be used from a remote client to ensure that any Add/Update/Delete will be stored to disk, via aClient.CallBackPut('Flush','','',dummy)

procedure CreateMissingTables(user_version: cardinal=0; Options: TSQLInitializeTableOptions=[]); override;
  Missing tables are created if they don’t exist yet for every TSQLRecord class of the Database Model
  - you must call explicitly this before having called StaticDataCreate()
  - all table description (even Unique feature) is retrieved from the Model
  - this method also create additional fields, if the TSQLRecord definition has been modified; only field adding is available, field renaming or field deleting are not allowed in the FrameWork (in such cases, you must create a new TSQLRecord type)

procedure DefinitionTo(Definition: TSynConnectionDefinition); override;
  Save the TSQLRestFullMemory properties into a persistent storage object
  - CreateFrom() will expect Definition.ServerName to store the FileName, and use binary storage if Definition.DatabaseName is not void

procedure DropDatabase; virtual;
  Clear all internal storage content

procedure LoadFromFile; virtual;
  Load the content from the specified file name
  - do nothing if file name was not assigned

procedure LoadFromStream(aStream: TStream); virtual;
  Load the content from the supplied resource
procedure UpdateToFile; virtual;
   Write any modification into file
   - do nothing if file name was not assigned

property BinaryFile: Boolean read fBinaryFile write fBinaryFile;
   Set if the file content is to be compressed binary, or standard JSON
   - it will use TSQLRestStorageInMemory LoadFromJSON/LoadFromBinary
     SaveToJSON/SaveToBinary methods for optimized storage

property FileName: TFileName read fFileName write fFileName;
   The file name used for data persistence

property Storage[aTable: TSQLRecordClass]: TSQLRestStorageInMemory read GetStorage;
   Direct access to the storage instances
   - you can then access to Storage[Table].Count and Storage[Table].Items[]

property Storages: TSQLRestStorageInMemoryDynArray read fStorage;
   Direct access to the storage instances
   - you can then access via Storage[TableIndex].Count and Items[]

TSQLRestServerRemoteDB = class(TSQLRestServer)
   A REST server using another TSQLRest instance for all its ORM process
   - this server will use an internal TSQLRest instance to handle all ORM operations (i.e. access to objects) - e.g. TSQLRestClient for remote access
   - it can be used e.g. to host some services on a stand-alone server, with all ORM and data access retrieved from another server: it will allow to easily implement a proxy architecture (for instance, as a DMZ for publishing services, but letting ORM process stay out of scope)
   - for per-table redirection, consider using the TSQLRestStorageRemote class via a call to the TSQLRestServer.RemoteDataCreate() method

constructor Create(aRemoteRest: TSQLRest; aHandleUserAuthentication: boolean=false); reintroduce; virtual;
   Initialize a REST server associated to a given TSQLRest instance
   - the specified TSQLRest will be used for all ORM and data process
   - you could use a TSQLRestClient or a TSQLRestServer instance
   - the supplied TSQLRest.Model will be used for TSQLRestServerRemoteDB
   - note that the TSQLRest instance won’t be freed - caller shall ensure that it will stay available at least until TSQLRestServerRemoteDB.Free

function AfterDeleteForceCoherency(TableIndex: integer; aID: TID): boolean; override;
   This method is called internally after any successful deletion to ensure relational database coherency
   - this overridden method will just return TRUE: in this remote access, true coherency will be performed on the ORM server side

property RemoteRest: TSQLRest read fRemoteRest;
   The remote ORM instance used for data persistence
   - may be a TSQLRestClient or a TSQLRestServer instance
TSQLRestClient = class(TSQLRest)

A generic REpresentational State Transfer (REST) client
- is RESTful (i.e. URI) remotely implemented (TSQLRestClientURI e.g.)
- is implemented for direct access to a database (TSQLRestClientDB e.g.)

function List(const Tables: array of TSQLRecordClass; const SQLSelect: RawUTF8 = 'RowID'; const SQLWhere: RawUTF8 = ''): TSQLTableJSON; virtual; abstract;

Retrieve a list of members as a TSQLTable
- implements REST GET collection
- default SQL statement is 'SELECT ID FROM TableName;' (i.e. retrieve the list of all ID of this collection members)
- optional SQLSelect parameter to change the returned fields as in 'SELECT SQLSelect FROM TableName,'
- optional SQLWhere parameter to change the search range or ORDER as in 'SELECT SQLSelect FROM TableName WHERE SQLWhere,'
- using inlined parameters via :(...): in SQLWhere is always a good idea
- for one TClass, you should better use TSQLRest.MultiFieldValues()

function ListFmt(const Tables: array of TSQLRecordClass; const SQLSelect, SQLWhereFormat: RawUTF8; const Args: array of const): TSQLTableJSON; overload;

Retrieve a list of members as a TSQLTable
- implements REST GET collection
- in this version, the WHERE clause can be created with the same format as FormatUTF8() function, replacing all '%' chars with Args[] values
- using inlined parameters via :(...) in SQLWhereFormat is always a good idea
- for one TClass, you should better use TSQLRest.MultiFieldValues()
- will call the List virtual method internally

function ListFmt(const Tables: array of TSQLRecordClass; const SQLSelect, SQLWhereFormat: RawUTF8; const Args, Bounds: array of const): TSQLTableJSON; overload;

Retrieve a list of members as a TSQLTable
- implements REST GET collection
- in this version, the WHERE clause can be created with the same format as FormatUTF8() function, replacing all '%' chars with Args[], and all '?' chars with Bounds[] (inlining them with :(...) and auto-quoting strings)
- example of use:
  Table := ListFmt([[TSQLRecord], 'Name', 'ID=?', [], [aID]]);
- for one TClass, you should better use TSQLRest.MultiFieldValues()
- will call the List virtual method internally

function Refresh(aID: TID; Value: TSQLRecord; var Refreshed: boolean): boolean;

Get a member from its ID
- implements REST GET collection
- URI is 'ModelRoot/TableName/TableID' with GET method
- returns true on server returned 200/HTTP_SUCCESS OK success, false on error
- set Refreshed to true if the content changed
function Retrieve(aID: TID; Value: TSQLRecord; ForUpdate: boolean=false): boolean;
override;

Get a member from its ID
- implements REST GET collection
- URI is 'ModelRoot/TableName/TableID' with GET method
- server must return Status 200/HTTP_SUCCESS OK on success
- if ForUpdate is true, the REST method is LOCK and not GET: it tries to lock the corresponding record, then retrieve its content; caller has to call UnLock() method after Value usage, to release the record

function ServiceContainer: TServiceContainer; override;

Access or initialize the internal IoC resolver, used for interface-based remote services, and more generally any Services.Resolve() call
- create and initialize the internal TServiceContainerClient if no service interface has been registered yet
- may be used to inject some dependencies, which are not interface-based remote services, but internal IoC, without the ServiceRegister() or ServiceDefine() methods - e.g. aRest.ServiceContainer.InjectResolver([TInfraRepoUserFactory.Create(aRest)],true);

function TransactionBegin(aTable: TSQLRecordClass; SessionID: cardinal=CONST_AUTHENTICATION_NOT_USED): boolean;
override;

Begin a transaction (calls REST BEGIN Member)
- by default, Client transaction will use here a pseudo session
- in aClient-Server environment with multiple Clients connected at the same time, you should better use BATCH process, specifying a positive AutomaticTransactionPerRow parameter to BatchStart()

function Update(Value: TSQLRecord; const CustomFields: TSQLFieldBits=[]; DoNotAutoComputeFields: boolean=false): boolean;
override;

Update a member
- implements REST PUT collection
- URI is 'ModelRoot/TableName/TableID' with PUT method
- server must return Status 200/HTTP_SUCCESS OK on success
- this overridden method will call BeforeUpdateEvent and also update BLOB fields, if any ForceBlobTransfert is set and CustomFields=[]

procedure Commit(SessionID: cardinal=CONST_AUTHENTICATION_NOT_USED; RaiseException: boolean=false); override;

End a transaction (calls REST END Member)
- by default, Client transaction will use here a pseudo session

procedure RollBack(SessionID: cardinal=CONST_AUTHENTICATION_NOT_USED); override;

Abort a transaction (calls REST ABORT Member)
- by default, Client transaction will use here a pseudo session
property ForceBlobTransfert: boolean
read GetForceBlobTransfert
write SetForceBlobTransfert;

If set to TRUE, all BLOB fields of all tables will be transferred between the Client and the remote Server
- i.e. Add() Update() will use Blob-related RESTful PUT/POST request
- i.e. Retrieve() will use Blob-related RESTful GET request
- note that the Refresh method won't handle BLOB fields, even if this property setting is set to TRUE
- by default, this property is set to FALSE, which setting will spare bandwidth and CPU
- this property is global to all tables of the model - you can also use ForceBlobTransfertTable[] to
force it for a particular table

property ForceBlobTransfertTable[aTable: TSQLRecordClass]: Boolean
read GetForceBlobTransfertTable
write SetForceBlobTransfertTable;

If set to TRUE for a specified table of the model, all BLOB fields of this tables will be transferred
between the Client and the remote Server
- i.e. Add() Update() will use BLOB-related RESTful PUT/POST request for this table
- i.e. Retrieve() will use BLOB-related RESTful GET request for this table
- note that the Refresh method won't handle BLOB fields, even if this property setting is set to TRUE
- by default, all items of this property are set to FALSE, which setting will spare bandwidth and
CPU
- this property is particular to a given tables of the model - you can also use ForceBlobTransfert
to force it for a all tables of this model

property OnRecordUpdate: TOnRecordUpdate
read fOnRecordUpdate
write fOnRecordUpdate;

This Event is called by Update() to let the client perform the record update (refresh associated
report e.g.)

property OnTableUpdate: TOnTableUpdate
read fOnTableUpdate
write fOnTableUpdate;

This Event is called by UpdateFromServer() to let the Client adapt to some rows update (for
Marked[] e.g.)

TSQLRestClientCallbackItem = record
Store information about registered interface callbacks

Factory: TInterfaceFactory;
/ information about the associated IInvokable

ID: integer;
The identifier of the callback, as sent to the server side
- computed from TSQLRestClientURICallsbacks.fCurrentID counter

Instance: pointer;
Weak pointer typecast to the associated IInvokable variable

ReleasedFromServer: boolean;
Set to TRUE if the instance was released from the server
**TSQLRestClientCallbacks** = class(TSynPersistentLock)

*Store the references to active interface callbacks on a REST Client*

- **Count**: integer;
  *How many callbacks are registered*

- **List**: array of TSQLRestClientCallbackItem;
  *List of registered interface callbacks*

- **Owner**: TSQLRestClientURI;
  *The associated REST instance*

**constructor** Create(aOwner: TSQLRestClientURI); reintroduce;

*Initialize the storage list*

**function** DoRegister(aInstance: pointer; aFactory: TInterfaceFactory): integer; overload;

*Register a callback event interface instance from a new computed ID*

**function** FindAndRelease(aID: integer): boolean;

*Find a matching entry*
  - will call FindIndex(aID) within Safe.Lock/Safe.Unlock
  - returns TRUE if aID was found and aInstance/aFactory set, FALSE otherwise

**function** FindEntry(var aItem: TSQLRestClientCallbackItem): boolean;

*Find a matching callback*
  - will call FindIndex(aItem.ID) within Safe.Lock/Safe.Unlock
  - returns TRUE if aItem.ID was found and aItem filled, FALSE otherwise

**function** FindIndex(aID: integer): integer;

*Find the index of the ID in the internal list*
  - warning: this method should be called within Safe.Lock/Safe.Unlock

**function** UnRegister(aInstance: pointer): boolean; overload;

*Delete all callback events from the internal list, as specified by its instance*
  - note that the same Invokable instance may be registered for several IDs

**procedure** DoRegister(aID: Integer; aInstance: pointer; aFactory: TInterfaceFactory); overload;

*Register a callback event interface instance from its supplied ID*

---

**TSQLRestClientURI** = class(TSQLRestClient)

*A generic REpresentationational State Transfer (REST) client with URI*

- URI are standard Collection/Member implemented as ModelRoot/TableName/TableID
- handle RESTful commands GET POST PUT DELETE LOCK UNLOCK

*Used for DI-2.1.1 (page 2543), DI-2.1.1.1 (page 2543), DI-2.1.1.2.2 (page 2544), DI-2.1.1.2.3 (page 2544), DI-2.1.1.2.4 (page 2545).*

**constructor** Create(aModel: TSQLModel); override;

*Initialize REST client instance*
**Synopse mORMot Framework**  
Software Architecture Design 1.18  
Date: September 16, 2020

---

destructor Destroy; override;

Release memory and close client connection
- also unlock all still locked records by this client

function BatchAdd(Value: TSQLRecord; SendData: boolean; ForceID: boolean=false; const CustomFields: TSQLFieldBits=[]): integer;

Create a new member in current BATCH sequence
- is a wrapper around TSQLRestBatch.Add() which will be stored in this TSQLRestClientURI instance - be aware that this won’t be thread safe

function BatchCount: integer;

Retrieve the current number of pending transactions in the BATCH sequence
- every call to BatchAdd/Update/Delete methods increases this count

function BatchDelete(ID: TID): integer; overload;

Delete a member in current BATCH sequence
- is a wrapper around TSQLRestBatch.Delete() which will be stored in this TSQLRestClientURI instance - be aware that this won’t be thread safe

function BatchDelete(Table: TSQLRecordClass; ID: TID): integer; overload;

Delete a member in current BATCH sequence
- is a wrapper around TSQLRestBatch.Delete() which will be stored in this TSQLRestClientURI instance - be aware that this won’t be thread safe

function BatchSend(var Results: TIDDynArray): integer; overload;

Execute a BATCH sequence started by BatchStart method
- send all pending BatchAdd/Update/Delete statements to the remote server
- URI is 'ModelRoot/TableName/0' with POST (or PUT) method
- will return the URI Status value, i.e. 200/HTTP_SUCCESS OK on success
- a dynamic array of integers will be created in Results, containing all ROWDID created for each BatchAdd call, 200 (=HTTP_SUCCESS) for all successful BatchUpdate/BatchDelete, or 0 on error
- any error during server-side process MUST be checked against Results[] (the main URI Status is 200 if about communication success, and won’t imply that all statements in the BATCH sequence were successful

function BatchStart(aTable: TSQLRecordClass; AutomaticTransactionPerRow: cardinal=0; Options: TSQLRestBatchOptions=[]): boolean; virtual;

Begin a BATCH sequence to speed up huge database change for a given table
- is a wrapper around TSQLRestBatch.Create() which will be stored in this TSQLRestClientURI instance - be aware that this won’t be thread-safe
- if you need a thread-safe "Unit Of Work" process, please use a private TSQLRestBatch instance and the overloaded TSQLRest.BatchSend() method
- call BatchStartAny() or set the aTable parameter to nil if you want to use any kind of TSQLRecord objects within the process, not a single one

function BatchStartAny(AutomaticTransactionPerRow: cardinal; Options: TSQLRestBatchOptions=[]): boolean;

Begin a BATCH sequence to speed up huge database change for any table
- will call the BatchStart() method with aTable = nil so that you may be able to use any kind of TSQLRecord class within the process
- is a wrapper around TSQLRestBatch.Create() which will be stored in this TSQLRestClientURI instance - be aware that this won’t be thread-safe
function BatchUpdate(Value: TSQLRecord; const CustomFields: TSQLFieldBits=[]; DoNotAutoComputeFields: boolean=false): integer;

   Update a member in current BATCH sequence
   - is a wrapper around TSQLRestBatch.Update() which will be stored in this TSQLRestClientURI instance - be aware that this won’t be thread safe
   - this method will call BeforeUpdateEvent before TSQLRestBatch.Update

function CallBack(method: TSQLURIMethod; const aMethodName,aSentData: RawUTF8; out aResponse: RawUTF8; aTable: TSQLRecordClass=nil; aID: TID=0; aResponseHead: PRawUTF8=nil): integer;

   Wrapper to the protected URI method to call a method on the server, using a ModelRoot/[TableName/[ID/]]MethodName RESTful with any kind of request
   - returns the HTTP error code (e.g. 200/HTTP_SUCCESS on success)
   - for GET/PUT methods, you should better use CallBackGet/CallBackPut

function CallBackGet(const aMethodName: RawUTF8; const aNameValueParameters: array of const; out aResponse: RawUTF8; aTable: TSQLRecordClass=nil; aID: TID=0; aResponseHead: PRawUTF8=nil): integer;

   Wrapper to the protected URI method to call a method on the server, using a ModelRoot/[TableName/[ID/]]MethodName RESTful GET request
   - returns the HTTP error code (e.g. 200/HTTP_SUCCESS on success)
   - this version will use a GET with supplied parameters (which will be encoded with the URL)

function CallBackGetResult(const aMethodName: RawUTF8; const aNameValueParameters: array of const; aTable: TSQLRecordClass=nil; aID: TID=0): RawUTF8;

   Wrapper to the protected URI method to call a method on the server, using a ModelRoot/[TableName/[ID/]]MethodName RESTful GET request
   - returns the UTF-8 decoded JSON result (server must reply with one "result":"value" JSON object)
   - this version will use a GET with supplied parameters (which will be encoded with the URL)

function CallBackPut(const aMethodName, aSentData: RawUTF8; out aResponse: RawUTF8; aTable: TSQLRecordClass=nil; aID: TID=0; aResponseHead: PRawUTF8=nil): integer;

   Wrapper to the protected URI method to call a method on the server, using a ModelRoot/[TableName/[ID/]]MethodName RESTful PUT request
   - returns the HTTP error code (e.g. 200/HTTP_SUCCESS on success)
   - this version will use a PUT with the supplied raw UTF-8 data

function ExecuteList(const Tables: array of TSQLRecordClass; const SQL: RawUTF8): TSQLTableJSON; override;

   Execute directly a SQL statement, expecting a list of results
   - URI is 'ModelRoot' with GET method, and SQL statement sent as UTF-8
   - return a result table on success, nil on failure

function List(const Tables: array of TSQLRecordClass; const SQLSelect: RawUTF8 = 'RowID'; const SQLWhere: RawUTF8 = ''): TSQLTableJSON; override;

   Retrieve a list of members as a TSQLTable
   - implements REST GET collection
   - URI is 'ModelRoot/TableName' with GET method
   - SQLSelect and SQLWhere are encoded as 'select=' and 'where=' URL parameters (using inlined parameters via {...}: in SQLWhere is always a good idea)
   - server must return Status 200/HTTP_SUCCESS OK on success
function ServerCacheFlush(aTable: TSQLRecordClass=nil; aID: TID=0): boolean;
virtual;

Send a flush command to the remote Server cache
- this method will remotely call the Cache.Flush() methods of the server instance, to force
cohesion of the data
- ServerCacheFlush() with no parameter will flush all stored JSON content
- ServerCacheFlush(aTable) will flush the cache for a given table
- ServerCacheFlush(aTable,aID) will flush the cache for a given record

function ServerInternalState: cardinal;

Ask the server for its current internal state revision counter
- this counter is incremented every time the database is modified
- the returned value is 0 if the database doesn't support this feature
- TSQLTable does compare this value with its internal one to check if its content must be
updated

function ServerRemoteLog(Sender: TTextWriter; Level: TSynLogInfo; const Text: RawUTF8): boolean; overload;
virtual;

Asynchronous call a 'RemoteLog' remote logging method on the server
- as implemented by mORMot's LogView tool in server mode
- to be used via ServerRemoteLogStart/ServerRemoteLogStop methods
- a dedicated background thread will run the transmission process without blocking the main
program execution, gathering log rows in chunks in case of high activity
- map TOneTextWriterEcho signature, so that you will be able to set e.g.:

function ServerRemoteLog(Level: TSynLogInfo; FormatMsg: PUTF8Char; const Args: array of const): boolean; overload;

Internal method able to emulate a call to TSynLog.Add.Log()
- will compute timestamp and event text, than call the overloaded ServerRemoteLog() method

function ServerTimestampSynchronize: boolean;

You can call this method to call the remote URI root/Timestamp
- this can be an handy way of testing the connection, since this method is always available, even
without authentication
- returns TRUE if the client time correction has been retrieved
- returns FALSE on any connection error - check LastErrorMessage and LastErrorException to find
out the exact connection error

function ServiceDefine(const aInterfaces: array of TGUID; aInstanceCreation: TServiceInstanceImplementation=sicSingle; const aContractExpected: RawUTF8=''): boolean; overload;

Register one or several Services on the client side via their interfaces
- this method expects the interface(s) to have been registered previously:
  TInterfaceFactory.RegisterInterFaces([TypeInfo(IMyInterface),...]);
function ServiceDefine(const aInterface: TGUID; aInstanceCreation: TServiceInstanceImplementation=sicSingle; const aContractExpected: RawUTF8=''; aIgnoreAnyException: boolean=true): TServiceFactoryClient; overload;

Register a Service on the client side via its interface
- this method expects the interface to have been registered previously:
  TInterfaceFactory.RegisterInterfaces([TypeInfo(IMyInterface),...]);

function ServiceDefineClientDriven(const aInterface: TGUID; out Obj; const aContractExpected: RawUTF8=''): boolean;

Register and retrieve the sicClientDriven Service instance
- this method expects the interface to have been registered previously:
  TInterfaceFactory.RegisterInterfaces([TypeInfo(IMyInterface),...]);

function ServiceDefineSharedAPI(const aInterface: TGUID; const aContractExpected: RawUTF8=SERVICE_CONTRACT_NONE_EXPECTED; aIgnoreAnyException: boolean=false): TServiceFactoryClient;

Register a sicShared Service instance communicating via JSON objects
- will force SERVICE_CONTRACT_NONE_EXPECTED, ParamsAsJSONObject=true and
  ResultAsJSONObjectWithoutResult=true
- may be used e.g. for accessing a sessionless public REST/JSON API, i.e.
  TSQLRestServer.ServiceDefine(...).ResultAsJSONObjectWithoutResult := true
- this method expects the interface to have been registered previously:
  TInterfaceFactory.RegisterInterfaces([TypeInfo(IMyInterface),...]);
- aIgnoreAnyException may be set to TRUE if the server is likely to not propose this service, and
  any exception is to be caught

function ServiceRegister(const aInterfaces: array of PTypeInfo; aInstanceCreation: TServiceInstanceImplementation=sicSingle; const aContractExpected: RawUTF8=''): boolean; overload; virtual;

Register one or several Services on the client side via their interfaces
- this methods expects a list of interfaces to be registered to the client (e.g. [TypeInfo(IMyInterface)])
- instance implementation pattern will be set by the appropriate parameter
- will return true on success, false if registration failed (e.g. if any of the supplied interfaces is not
  correct or is not available on the server)
- that is, server side will be called to check for the availability of each interface
- you can specify an optional custom contract for the first interface

function ServiceRegister(aInterface: PTypeInfo; aInstanceCreation: TServiceInstanceImplementation=sicSingle; const aContractExpected: RawUTF8=''): TServiceFactory; overload;

Register a Service on the client side via its interface
- this methods expects one interface to be registered to the client, as
  Client.ServiceRegister(TypeInfo(IMyInterface),sicShared);
- instance implementation pattern will be set by the appropriate parameter
- will return the corresponding fake class factory on success, nil if registration failed (e.g. if any
  of supplied interfaces is not correct or is not available on the server)
- that is, server side will be called to check for the availability of each interface
- you can specify an optional custom contract for the first interface
function ServiceRegisterClientDriven(aInterface: PTypeInfo; out Obj; const aContractExpected: RawUTF8=''): boolean; overload;

Register and retrieve the sicClientDriven Service instance
- will return TRUE on success, filling Obj output variable with the corresponding interface instance
- will return FALSE on error

function ServiceRetrieveAssociated(const aServiceName: RawUTF8; out URI: TSQLRestServerURIDynArray): boolean; overload;

Return all REST server URI associated to this client, for a given service name, the latest registered in first position
- will lookup for the Interface name without the initial 'I', e.g. 'Calculator' for ICalculator - warning: research is case-sensitive
- this methods is the reverse from ServicePublishOwnInterfaces: it allows to guess an associated REST server which may implement a given service

function ServiceRetrieveAssociated(const aInterface: TGUID; out URI: TSQLRestServerURIDynArray): boolean; overload;

Return all REST server URI associated to this client, for a given service
- here the service is specified as its TGUID, e.g. IMyInterface
- this method expects the interface to have been registered previously:
  TInterfaceFactory.RegisterInterfaces([TypeInfo(IMyInterface),...]);
- the URI[] output array contains the matching server URIs, the latest registered in first position
- this methods is the reverse from ServicePublishOwnInterfaces: it allows to guess an associated REST server which may implement a given service

function SetUser(const aUserName, aPassword: RawUTF8; aHashedPassword: Boolean=false): boolean;

Authenticate an User to the current connected Server
- will call the ModelRoot/Auth service, i.e. call TSQLRestServer.Auth() published method to create a session for this user, with our secure TSQLRestServerAuthenticationDefault authentication scheme
- returns true on success
- calling this method is optional, depending on your user right policy: your Server need to handle authentication
- if saoUserByLogonOrID is defined in the server Options, aUserName may be a TSQLAuthUser.ID integer value and not a TSQLAuthUser.LogonName
- on success, the SessionUser property map the logged user session on the server side
- if aHashedPassword is TRUE, the aPassword parameter is expected to contain the already-hashed value, just as stored in PasswordHashHexa (i.e. SHA256('salt'+Value) as in TSQLAuthUser.SetPasswordPlain method)
- if SSPIAUTH conditional is defined, and aUserName='', a Windows authentication will be performed via TSQLRestServerAuthenticationSSPI - in this case, aPassword will contain the SPN domain for Kerberos (otherwise NTLM will be used), and table TSQLAuthUser shall contain an entry for the logged Windows user, with the LoginName in form 'DomainName\UserName'
- you can directly create the class method ClientSetUser() of a given TSQLRestServerAuthentication inherited class, if neither TSQLRestServerAuthenticationDefault nor TSQLRestServerAuthenticationSSPI match your need
function TransactionBegin(aTable: TSQLRecordClass; SessionID: cardinal=1): boolean; override;

Begin a transaction
- implements REST BEGIN collection
- in a Client-Server environment with multiple Clients connected at the same time, you should better use BATCH process, specifying a positive AutomaticTransactionPerRow parameter to BatchStart()
- may be used to speed up some SQL statements as Add/Update/Delete methods
- must be ended with Commit on success
- in the current implementation, the aTable parameter is not used yet
- must be aborted with Rollback if any SQL statement failed
- return true if no transaction is active, false otherwise

if Client.TransactionBegin(TSQLRecordPeopleObject) then
try
    // .... modify the database content, raise exceptions on error
    Client.Commit;
except
    Client.Rollback; // in case of error
end;

- you may use the dedicated TransactionBeginRetry() method in case of potential Client concurrent access

function TransactionBeginRetry(aTable: TSQLRecordClass; Retries: integer=10): boolean;

Begin a transaction
- implements REST BEGIN collection
- in a Client-Server environment with multiple Clients connected at the same time, you should better use BATCH process, specifying a positive AutomaticTransactionPerRow parameter to BatchStart()
- this version retries a TransactionBegin() to be successful within a supplied number of times
- will retry every 100 ms for "Retries" times (excluding the connection time in this 100 ms time period
- default is to retry 10 times, i.e. within 2 second timeout
- in the current implementation, the aTable parameter is not used yet
- typical usage should be for instance:

if Client.TransactionBeginRetry(TSQLRecordPeopleObject, 20) then
try
    // .... modify the database content, raise exceptions on error
    Client.Commit;
except
    Client.Rollback; // in case of error
end;

function Unlock(Table: TSQLRecordClass; aID: TID): boolean; override;

Unlock the corresponding record
- URI is 'ModelRoot/TableName/TableID' with UNLOCK method
- returns true on success
function UpdateFromServer(const Data: array of TObject; out Refreshed: boolean; PCurrentRow: PInteger=nil): boolean;
Check if the data may have changed of the server for this objects, and update it if possible
- only working types are TSQLTableJSON and TSQLRecord descendants
- make use of the InternalState function to check the data content revision
- return true if Data is updated successfully, or false on any error during data retrieval from server (e.g. if the TSQLRecord has been deleted)
- if Data contains only one TSQLTableJSON, PCurrentRow can point to the current selected row of this table, in order to refresh its value
- use this method to refresh the client UI, e.g. via a timer

function URI(const url, method: RawUTF8; Resp: PRawUTF8=nil; Head: PRawUTF8=nil; SendData: PRawUTF8=nil): Int64Rec;
Method calling the remote Server via a RESTful command
- calls the InternalURI abstract method, which should be overridden with a local, piped or HTTP/1.1 provider
- this method will add sign the url with the appropriate digital signature according to the current SessionUser property
- this method will retry the connection in case of authentication failure (i.e. if the session was closed by the remote server, for any reason - mostly a time out) if the OnAuthenticationFailed event handler is set

procedure BatchAbort;
Abort a BATCH sequence started by BatchStart method
- in short, nothing is sent to the remote server, and current BATCH sequence is closed
- will Free the TSQLRestBatch stored in this TSQLRestClientURI instance

procedure CallbackNonBlockingSetHeader(out Header: RawUTF8); virtual;
To be called before CallBack() if the client could ignore the answer
- do nothing by default, but overridden e.g. in TSQLHttpClientWebsockets

procedure Commit(SessionID: cardinal=CONST_AUTHENTICATION_NOT_USED; RaiseException: boolean=false); override;
End a transaction
- implements REST END collection
- write all pending SQL statements to the disk

procedure DefinitionTo(Definition: TSynConnectionDefinition); override;
Save the TSQLRestClientURI properties into a persistent storage object
- CreateFrom() will expect Definition.UserName/Password to store the credentials which will be used by SetUser()

procedure RollBack(SessionID: cardinal=CONST_AUTHENTICATION_NOT_USED); override;
Abort a transaction
- implements REST ABORT collection
- restore the previous state of the database, before the call to TransactionBegin
procedure ServerRemoteLogStart(aLogClass: TSynLogClass; aClientOwnedByFamily: boolean);

Start to send all logs to the server 'RemoteLog' method-based service
- will associate the EchoCustom callback of the running log class to the ServerRemoteLog() method
- if aClientOwnedByFamily is TRUE, this TSQLRestClientURI instance lifetime will be managed by TSynLogFamily - which is mostly wished
- if aClientOwnedByFamily is FALSE, you should manage this instance life time, and may call ServerRemoteLogStop to stop remote logging
- warning: current implementation will disable all logging for this TSQLRestClientURI instance, to avoid any potential concern (e.g. for multi-threaded process, or in case of communication error): you should therefore use this TSQLRestClientURI connection only for the remote log server, e.g. via TSQHTTPClientGeneric.CreateForRemoteLogging() - do not call ServerRemoteLogStart() from a high-level business client!

procedure ServerRemoteLogStop;

Stop sending all logs to the server 'RemoteLog' method-based service
- do nothing if aClientOwnedByFamily was TRUE for ServerRemoteLogStart

class procedure ServiceNotificationMethodExecute(var Msg : TMessage);

Event to be triggered when a WM_* message is received from the internal asynchronous notification system, to run the callback in the main UI thread
- WM_* message identifier should have been set e.g. via the associated ServiceNotificationMethodViaMessages(Form.Handle,WM_USER)
- message will be sent for any interface-based service method callback which expects no result (i.e. no out parameter nor function result), so is safely handled as asynchronous notification
- is defines as a class procedure, since the underlying TSQLRestClientURI instance has no impact here: a single WM_* handler is enough for several TSQLRestClientURI instances

procedure ServiceNotificationMethodViaMessages(hWnd: HWND; Msg: UINT);

Set a HWND/WM_* pair to let interface-based services notification callbacks be processed safely in the main UI thread, via Windows messages
- by default callbacks are executed in the transmission thread, e.g. the WebSockets client thread: using VCL Synchronize() method may trigger some unexpected race conditions, e.g. when asynchronous notifications are received during a blocking REST command - this message-based mechanism will allow safe and easy notification for any VCL client application
- the associated ServiceNotificationMethodExecute() method shall be called in the client HWND TForm for the defined WM_* message

procedure ServicePublishOwnInterfaces(OwnServer: TSQLRestServer);

Allow to notify a server the services this client may be actually capable
- when this client will connect to a remote server to access its services, it will register its own services, supplying its TSQLRestServer instance, and its corresponding public URI, within its '_contract_' internal call
- it will allow automatic service discovery of Peer To Peer Servers, without the need of an actual centralized SOA catalog service: any client could retrieve an associated REST server for a given service, via the ServiceRetrieveAssociated method
procedure SessionClose;

Clear session and call the /auth service on the server to notify shutdown
- is called by Destroy and SetUser/ClientSetUser methods, so you should not have usually to call
this method directly

property ComputeSignature: TSQLRestServerAuthenticationSignedURIComputeSignature read fComputeSignature write fComputeSignature;

Customize the session_signature signing algorithm with a specific function
- will be used by TSQLRestServerAuthenticationSignedURI classes, e.g.
TSQLRestServerAuthenticationDefault instead of the algorithm specified by the server at session
handshake

property LastErrorCode: integer read fLastErrorCode;

Low-level error code, as returned by server
- check this value about HTTP_* constants
- HTTP_SUCCESS or HTTP_CREATED mean no error
- otherwise, check LastErrorMessage property for additional information
- this property value will record status codes returned by URI() method

property LastErrorException: ExceptClass read fLastErrorException;

Low-level exception class, if any
- will record any Exception class raised within URI() method
- contains nil if URI() execution did not raise any exception (which is the most expected behavior,
since server-side errors are trapped into LastErrorCode/LastErrorMessage properties

property LastErrorMessage: RawUTF8 read fLastErrorMessage;

Low-level error message, as returned by server
- this property value will record content returned by URI() method in case of an error, or " if
LastErrorCode is HTTP_SUCCESS or HTTP_CREATED

property MaximumAuthenticationRetry: Integer read fMaximumAuthenticationRetry write fMaximumAuthenticationRetry;

Maximum additional retry occurrence
- default is 1, i.e. will retry once
- set OnAuthenticationFailed to nil in order to avoid any retry

property OnAuthenticationFailed: TOnAuthenticationFailed read fOnAuthenticationFailed write fOnAuthenticationFailed;

This Event is called in case of remote authentication failure
- client software can ask the user to enter a password and user name
- if no event is specified, the URI() method will return directly an HTTP_FORBIDDEN "403 Forbidden" error code

property OnFailed: TOnClientFailed read fOnFailed write fOnFailed;

This Event is called if URI() was not successful
- the callback will have all needed information
- e.g. Call^.OutStatus=HTTP_NOTIMPLEMENTED indicates a broken connection
property OnIdle: TOnIdleSymBackgroundThread read fOnIdle write fOnIdle;

Set a callback event to be executed in loop during remote blocking process, e.g. to refresh the UI during a somewhat long request
- if not set, the request will be executed in the current thread, so may block the User Interface
- you can assign a callback to this property, calling for instance Application.ProcessMessages, to execute the remote request in a background thread, but let the UI still be reactive: the TLoginForm.OnIdleProcess and OnIdleProcessForm methods of mORMotUILogin.pas will match this property expectations

property OnIdleBackgroundThreadActive: Boolean read GetOnIdleBackgroundThreadActive;

TRUE if the background thread is active, and OnIdle event is called during process
- to be used e.g. to ensure no re-entrance from User Interface messages

property OnSetUser: TOnRestClientNotify read fOnSetUser write fOnSetUser;

This Event is called when a user is authenticated
- is called always, on each TSQLRestClientURI.SetUser call
- you can check the Sender.SessionUser property pointing to the current authenticated user, or nil if authentication failed
- could be used to refresh the User Interface layout according to current authenticated user rights, or to subscribe to some services via callbacks

property RetryOnceOnTimeout: Boolean read fRetryOnceOnTimeout write fRetryOnceOnTimeout;

If the client shall retry once in case of "408 REQUEST TIMEOUT" error

property SessionHeartbeatSeconds: integer read fSessionHeartbeatSeconds write SetSessionHeartbeatSeconds;

Frequency of Callback/_ping_ calls to maintain session and services
- will be used to call SessionRenewEvent at the specified period, so that the session and all sicClientDriven instances will be maintained on the server side as long as the client connection stands
- equals half SessionServerTimeout or 25 minutes (if lower) by default - 25 minutes matches the default service timeout of 30 minutes
- you may set 0 to disable this SOA-level heartbeat feature

property SessionHttpHeader: RawUTF8 read fSessionHttpHeader write fSessionHttpHeader;

Access to the low-level HTTP header used for authentication
- you can force here your own header, e.g. a JWT as authentication bearer or as in TSQLRestServerAuthenticationHttpAbstract.ClientSetUserHttpOnlyUser

property SessionID: cardinal read fSessionID;

The current session ID as set after a sucessful SetUser() method call
- equals 0 (CONST_AUTHENTICATION_SESSION_NOT_STARTED) if the session is not started yet - i.e. if SetUser() call failed
- equals 1 (CONST_AUTHENTICATION_NOT_USED) if authentication mode is not enabled - i.e. after a fresh Create() without SetUser() call

property SessionServer: RawUTF8 read fSessionServer;

The remote server executable name, as retrieved after a SetUser() success
property SessionServerTimeout: integer read fSessionServerTimeout;
  The remote server session timeout in minutes, as retrieved after a SetUser() success
  - will be used to set SessionHeartbeatSeconds default

property SessionUser: TSQLAuthUser read fSessionUser;
  The current user as set by SetUser() method
  - contains nil if no User is currently authenticated
  - once authenticated, a TSQLAuthUser instance is set, with its ID, LogonName, DisplayName,
    PasswordHashHexa and GroupRights (filled with a TSQLAuthGroup ID casted as a pointer)
  properties - you can retrieve any optional binary data associated with this user via
    RetrieveBlobFields()

property SessionVersion: RawUTF8 read fSessionVersion;
  The remote server version, as retrieved after a SetUser() success

TSQLRestClientURIDll = class(TSQLRestClientURI)
  Rest client with remote access to a server through a dll
  - use only one TURIMapRequest function for the whole communication
  - the data is stored in Global system memory, and freed by GlobalFree()
  Used for DI-2.1.1.2.1 (page 2544).

constructor Create(aModel: TSQLModel; const DllName: TFileName); reintroduce;
  overload;
  Connect to a server contained in a shared library
  - this dll must contain at least a URIRequest entry
  - raise an exception if the shared library is not found or invalid
  Used for DI-2.1.1.2.1 (page 2544).

constructor Create(aModel: TSQLModel; aRequest: TURIMapRequest); reintroduce;
  overload;
  Connect to a server from a remote function
  Used for DI-2.1.1.2.1 (page 2544).

destructor Destroy; override;
  Release memory and handles

TSQLRestClientRedirect = class(TSQLRestClientURI)
  Rest client with redirection to another TSQLRest instance

constructor Create(aRedirected: TSQLRest); reintroduce; overload;
  Will pass all client commands to the supplied TSQLRest instance
  - aRedirected is expected to be either a TSQLRestClientURI or a TSQLRestServer
  - will make a copy of the aRedirected.Model, and own it

constructor Create(aModel: TSQLModel); overload; override;
  Prepare the redirection, to be enabled later via RedirectTo()
  - the supplied aModel instance will be owned by this class
**TSQLRestClientURIMessage** = class(TSQLRestClientURI)

*Rest client with remote access to a server through Windows messages*
- use only one TURIMapRequest function for the whole communication
- the data is sent and received by using the standard and fast WM_COPYDATA message
- named pipes seems to be somewhat better for bigger messages under XP
- this class is thread-safe, since its URI() method is protected by a lock

*Used for Di-2.1.1.2.3 (page 2544).*

**constructor** CreateOwned(aRedirected: TSQLRestServer);

*reintroduce;*

*Will pass all client commands to the supplied TSQLRestServer instance*
- aRedirected will be owned by this TSQLRestClientRedirect

**procedure** RedirectTo(aRedirected: TSQLRest);

*Allows to change redirection to a client on the fly*
- if aRedirected is nil, redirection will be disabled and any URI() call will return an HTTP_GATEWAYTIMEOUT 504 error status

**constructor** Create(aModel: TSQLModel; const ServerWindowName, ClientWindowName: string; TimeOutMS: cardinal);

*reintroduce; overload;*

*Connect to a server from its window name*
- ServerWindowName is of UnicodeString type since Delphi 2009 (direct use of FindWindow()=FindWindowW() Win32 API)
- this version will instantiate and create a Client Window from a Window Name, by using low level Win32 API: therefore, the Forms unit is not needed with this constructor (save some KB)

*Used for Di-2.1.1.2.3 (page 2544).*

**constructor** Create(aModel: TSQLModel; const ServerWindowName: string; ClientWindow: HWND; TimeOutMS: cardinal);

*reintroduce; overload;*

*Connect to a server from its window name*
- ServerWindowName is of UnicodeString type since Delphi 2009 (direct use of FindWindow()=FindWindowW() Win32 API)
- this version must supply a Client Window handle

*Used for Di-2.1.1.2.3 (page 2544).*

**destructor** Destroy;

*override;*

*Release the internal Window class created, if any*

**procedure** DefinitionTo(Definition: TSynConnectionDefinition);

*override;*

*Save the TSQLRestClientURIMessage properties into a persistent storage object*
- CreateFrom() will expect Definition.ServerName to store the ServerWindowName, and Definition.DatabaseName to be the ClientWindowName

**procedure** WMCopyData(var Msg : TWMCopyData);

*message WM_COPYDATA;*

*Event to be triggered when a WM_COPYDATA message is received from the server*
- to be called by the corresponding "message WM_COPYDATA;" method in the client TForm instance

Define if the client will process the Windows Messages loop
- set to TRUE if the client is used outside the main GUI application thread

TSQLRestClientURINamedPipe = class(TSQLRestClientURI)
Rest client with remote access to a server through a Named Pipe
- named pipe is fast and optimized under Windows
- can be accessed locally or remotely
- this class is thread-safe, since its URI() method is protected by a lock

Used for DI-2.1.1.2.2 (page 2544).

constructor Create(aModel: TSQLModel; const ApplicationName: TFileName);
reintroduce;

Connect to a server contained in a running application
- the server must have been declared by a previous
  TSQLRestServer.ExportServer(ApplicationName) call with ApplicationName as user-defined
  server identifier ('DBSERVER' e.g.)
- ApplicationName is of UnicodeString type since Delphi 2009 (direct use of Wide Win32 API
  version)
- this server identifier is appended to '\\pipe\mORMot_\' to obtain the full pipe name to
  connect to ('\\pipe\mORMot__DBSERVER' e.g.)
- this server identifier may also contain a remote computer name, and must be fully qualified
  ('\\ServerName\pipe\ApplicationName' e.g.)
- raise an exception if the server is not running or invalid

Used for DI-2.1.1.2.2 (page 2544).

procedure DefinitionTo(Definition: TSynConnectionDefinition); override;
Save the TSQLRestClientURIMessage properties into a persistent storage object
- CreateFrom() will expect Definition.ServerName to store the expected ApplicationName

TSynValidateRest = class(TSynValidate)
Will define a validation to be applied to a TSQLRecord field, using if necessary an associated
TSQLRest instance and a TSQLRecord class
- a typical usage is to validate a value to be unique in the table (implemented in the
  TSynValidateUniqueField class)
- the optional associated parameters are to be supplied JSON-encoded
- ProcessRest and ProcessRec properties will be filled before Validate method call by
  TSQLRecord.Validate()

property ProcessRec: TSQLRecord read fProcessRec;
The associated TSQLRecord instance
- this value is updated by Validate with the current TSQLRecord instance to be validated
- it can be used in the overridden DoValidate method
property ProcessRest: TSQLRest read fProcessRest;

The associated TSQLRest instance
- this value is updated by Validate with the current TSQLRest used for the validation
- it can be used in the overridden DoValidate method

TSynValidateUniqueField = class(TSynValidateRest)
Will define a validation for a TSQLRecord Unique text field
- this class will handle only textual fields, not numeric values
- it will check that the field value is not void
- it will check that the field value is not a duplicate

TSynValidateUniqueFields = class(TSynValidateRest)
Will define an unicity validation for a set of TSQLRecord text fields
- field names should be specified as CSV in the JSON "FieldNames" property in the constructor, or
  the Parameters field, e.g. like
  TSQLSampleRecord.AddFilterOrValidate('propA',
     TSynValidateUniqueFields.Create('{{"FieldNames":"propA,propB"}}'));
- this class will handle only textual fields, not numeric values
- it will check that the field values are not a duplicate

property FieldNames: TRawUTF8DynArray read fFieldNames;
The validated field names

TSQLVirtualTablePreparedConstraint = packed record
A WHERE constraint as set by the TSQLVirtualTable.Prepare() method

Column: integer;
   Column on left-hand side of constraint
   - The first column of the virtual table is column 0
   - The RowID of the virtual table is column -1
   - Hidden columns are counted when determining the column index
   - if this field contains VIRTUAL_TABLE_IGNORE_COLUMN (-2), TSQLVirtualTable. Prepare()
     should ignore this entry

OmitCheck: boolean;
   If true, the constraint is assumed to be fully handled by the virtual table and is not checked again
   by SQLite
   - By default (OmitCheck=false), the SQLite core double checks all constraints on each row of the
     virtual table that it receives
   - TSQLVirtualTable.Prepare() can set this property to true

Operation: TCompareOperator;
   Constraint operator
   - MATCH keyword is parsed into soBeginWith, and should be handled as soBeginWith,
     soContains or soSoundsLike* according to the effective expression text value ('text*', '%text'...)

mORMot.pas unit - Rev. 1.18
Value: TSQLVar;

The associated expression
- TSQLVirtualTable.Prepare() must set Value.VType to not svtUnknown (e.g. to svtNull), if an
expression is expected at vt_BestIndex() call
- TSQLVirtualTableCursor.Search() will receive an expression value, to be retrieved e.g. via
sqlite3_value_*() functions

TSQLVirtualTablePreparedOrderBy = record
An ORDER BY clause as set by the TSQLVirtualTable.Prepare() method
- warning: this structure should match exactly TSQLite3IndexOrderBy as defined in SynSQLite3

Column: Integer;

Column number
- The first column of the virtual table is column 0
- The RowID of the virtual table is column -1
- Hidden columns are counted when determining the column index.

Desc: boolean;

True for DESCending order, false for ASCending order.

TSQLVirtualTablePrepared = object(TObject)
The WHERE and ORDER BY statements as set by TSQLVirtualTable.Prepare
- Where[] and OrderBy[] are fixed sized arrays, for fast and easy code

EstimatedCost: TSQLVirtualTablePreparedCost;

Estimated cost of using this prepared index
- SQLite uses this value to make a choice between several calls to the TSQLVirtualTable.Prepare()
method with several expressions

EstimatedRows: Int64;

Estimated number of rows of using this prepared index
- does make sense only if EstimatedCost=costFullScan
- SQLite uses this value to make a choice between several calls to the TSQLVirtualTable.Prepare()
method with several expressions
- is used only starting with SQLite 3.8.2

OmitOrderBy: boolean;

If true, the ORDER BY statement is assumed to be fully handled by the virtual table and is not
checked again by SQLite
- By default (OmitOrderBy=false), the SQLite core sort all rows of the virtual table that it receives
according in order

OrderBy: array[0..MAX_SQLFIELDS-1] of TSQLVirtualTablePreparedOrderBy;
ORDER BY statement parameters

OrderByCount: integer;

Number of ORDER BY statement parameters in OrderBy[]

Where: array[0..MAX_SQLFIELDS-1] of TSQLVirtualTablePreparedConstraint;
WHERE statement parameters, in TSQLVirtualTableCursor.Search() order
WhereCount: integer;
Number of WHERE statement parameters in Where[] array

function IsWhereIDEquals(CalledFromPrepare: Boolean): boolean;
Returns TRUE if there is only one ID=? statement in this search

function IsWhereOneFieldEquals: boolean;
Returns TRUE if there is only one FieldName=? statement in this search

TVirtualTableModuleProperties = record
Used to store and handle the main specifications of a TSQLVirtualTableModule
CursorClass: TSQLVirtualTableCursorClass;
The associated cursor class
Features: TSQLVirtualTableFeatures;
A set of features of a Virtual Table
FileExtension: TFileName;
Can be used to customize the extension of the filename
- the '.' is not to be included
RecordClass: TSQLRecordClass;
The associated TSQLRecord class
- used to retrieve the field structure with all collations
StaticClass: TSQLRestStorageClass;
The associated TSQLRestStorage class used for storage
- is e.g. TSQLRestStorageInMemory for TSQLVirtualTableJSON, TSQLRestStorageExternal for TSQLVirtualTableExternal, or nil for TSQLVirtualTableLog

TSQLVirtualTableModule = class(TObject)
Parent class able to define a Virtual Table module
- in order to implement a new Virtual Table type, you'll have to define a so called "Module" to handle the fields and data access and an associated TSQLVirtualTableCursorClass for handling the SELECT statements
- for our framework, the SQLite3 unit will inherit from this class to define a TSQLVirtualTableModuleSQLite3 class, which will register the associated virtual table definition into a SQLite3 connection, on the server side
- children should override abstract methods in order to implement the association with the database engine itself
constructor Create(aTableClass: TSQLVirtualTableClass; aServer: TSQLRestServer);
virtual;
Create the Virtual Table instance according to the supplied class
- inherited constructors may register the Virtual Table to the specified database connection
function FileName(const aTableName: RawUTF8): TFileName; virtual;
Retrieve the file name to be used for a specific Virtual Table
- returns by default a file located in the executable folder, with the table name as file name, and module name as extension
property CursorClass: TSQLVirtualTableCursorClass read fFeatures.CursorClass;
The associated virtual table cursor class

property Features: TSQLVirtualTableFeatures read fFeatures.Features;
The Virtual Table module features

property FileExtension: TFileName read fFeatures.FileExtension;
The extension of the filename (without any left '.')

property FilePath: TFileName read fFilePath write fFilePath;
The full path to be used for the filename
- is '' by default, i.e. will use the executable path
- you can specify here a custom path, which will be used by the FileName method to retrieve the .json/.data full file

property ModuleName: RawUTF8 read fModuleName;
The corresponding module name

property RecordClass: TSQLRecordClass read fFeatures.RecordClass;
The associated TSQLRecord class
- is mostly nil, e.g. for TSQLVirtualTableJSON
- used to retrieve the field structure for TSQLVirtualTableLog e.g.

property Server: TSQLRestServer read fServer;
The associated Server instance
- may be nil, in case of direct access to the virtual table

property StaticClass: TSQLRestStorageClass read fFeatures.StaticClass;
The associated TSQLRestStorage class used for storage
- e.g. returns TSQLRestStorageInMemory for TSQLVirtualTableJSON, or TSQLRestStorageExternal for TSQLVirtualTableExternal, or either nil for TSQLVirtualTableLog

property TableClass: TSQLVirtualTableClass read fTableClass;
The associated virtual table class

TSQLVirtualTable = class(TObject)
Abstract class able to access a Virtual Table content
- override the Prepare/Structure abstract virtual methods for reading access to the virtual table content
- you can optionally override Drop/Delete/Insert/Update/Rename/Transaction virtual methods to allow content writing to the virtual table
- the same virtual table mechanism can be used with several database module, with diverse database engines
**constructor** Create(aModule: TSQLVirtualTableModule; const aTableName: RawUTF8; FieldCount: integer; Fields: PPUTF8CharArray); virtual;

Create the virtual table access instance
- the created instance will be released when the virtual table will be disconnected from the DB connection (e.g. xDisconnect method for SQLite3)
- shall raise an exception in case of invalid parameters (e.g. if the supplied module is not associated to a TSQLRestServer instance)
- aTableName will be checked against the current aModule.Server.Model to retrieve the corresponding TSQLRecordVirtualTableAutoID class and create any associated Static: TSQLRestStorage instance

**destructor** Destroy; override;

Release the associated memory, especially the Static instance

**function** Delete(aRowID: Int64): boolean; virtual;

Called to delete a virtual table row
- should return true on success, false otherwise
- does nothing by default, and returns false, i.e. always fails

**function** Drop: boolean; virtual;

Called when a DROP TABLE statement is executed against the virtual table
- should return true on success, false otherwise
- does nothing by default, and returns false, i.e. always fails

**function** Insert(aRowID: Int64; var Values: TSQLVarDynArray; out insertedRowID: Int64): boolean; virtual;

Called to insert a virtual table row content from an array of TSQLVar
- should return true on success, false otherwise
- should return the just created row ID in insertedRowID on success
- does nothing by default, and returns false, i.e. always fails

**class function** ModuleName: RawUTF8;

Retrieve the corresponding module name
- will use the class name, trimming any T/TSQL/TSQLVirtual/TSQLVirtualTable*
- when the class is instanciated, it will be faster to retrieve the same value via Module.ModuleName

**function** Prepare(var Prepared: TSQLVirtualTablePrepared): boolean; virtual;

Called to determine the best way to access the virtual table
- will prepare the request for TSQLVirtualTableCursor.Search()
- in Where[], Expr must be set to not 0 if needed for Search method, and OmitCheck to true if double check is not necessary
- OmitOrderBy must be set to true if double sort is not necessary
- EstimatedCost and EstimatedRows should receive the estimated cost
- default implementation will let the DB engine perform the search, and prepare for ID=? statement if vtWhereIDPrepared was set

**function** Rename(const NewName: RawUTF8): boolean; virtual;

Called to rename the virtual table
- by default, returns false, i.e. always fails
function Structure: RawUTF8; virtual;

Should retrieve the format (the names and datatypes of the columns) of the virtual table, as expected by sqlite3_declare_vtab()
- default implementation is to retrieve the structure for the associated Module.RecordClass property (as set by GetTableModuleProperties) or the Static.StoredClass: in both cases, column numbering will follow the TSQLRecord published field order (TSQLRecord.RecordProps.Fields[])

class function StructureFromClass(aClass: TSQLRecordClass; const aTableName: RawUTF8): RawUTF8;

A generic method to get a 'CREATE TABLE' structure from a supplied TSQLRecord class
- is called e.g. by the Structure method

function Transaction(aState: TSQLVirtualTableTransaction; aSavePoint: integer): boolean; virtual;

Called to begin a transaction to the virtual table row
- do nothing by default, and returns false in case of RollBack/RollBackTo
- aSavePoint is used for vttSavePoint, vttRelease and vttRollBackTo only
- note that if you don’t nest your writing within a transaction, SQLite will call vttCommit for each INSERT/UPDATE/DELETE, just like a regular SQLite database - it could make bad written code slow even with Virtual Tables

function Update(oldRowID, newRowID: Int64; var Values: TSQLVarDynArray): boolean; virtual;

Called to update a virtual table row content from an array of TSQLVar
- should return true on success, false otherwise
- does nothing by default, and returns false, i.e. always fails

class procedure GetTableModuleProperties( var aProperties: TVirtualTableModuleProperties); virtual; abstract;

Should return the main specifications of the associated TSQLVirtualTableModule

property Module: TSQLVirtualTableModule read fModule;

The associated Virtual Table module

property Static: TSQLRest read fStatic;

The associated virtual table storage instance
- can be e.g. a TSQLRestStorageInMemory for TSQLVirtualTableJSON, or a TSQLRestStorageExternal for TSQLVirtualTableExternal, or nil for TSQLVirtualTableLog

property StaticStorage: TSQLRestStorage read fStaticStorage;

The associated virtual table storage instance, if is a TSQLRestStorage

property StaticTable: TSQLRecordClass read fStaticTable;

The associated virtual table storage table

property StaticTableIndex: integer read fStaticTableIndex;

The associated virtual table storage index in its Model.Tables[] array

property TableName: RawUTF8 read fTableName;

The name of the Virtual Table, as specified following the TABLE keyword in the CREATE VIRTUAL TABLE statement
TSQVirtualTableCursor = class(TObject)

Abstract class able to define a Virtual Table cursor
- override the Search/HasData/Column/Next abstract virtual methods to implement the search process

constructor Create(aTable: TSQVirtualTable); virtual;
Create the cursor instance
- it will be destroyed when by the DB engine (e.g. via xClose in SQLite3)

function Column(aColumn: integer; var aResult: TSQLVar): boolean; virtual; abstract;
Called to retrieve a column value of the current data row into a TSQLVar
- if aColumn=-1, should return the row ID as varInt64 into aResult
- should return false in case of an error, true on success

function HasData: boolean; virtual; abstract;
Called after Search() to check if there is data to be retrieved
- should return false if reached the end of matching data

function Next: boolean; virtual; abstract;
Called to go to the next row of matching data
- should return false on low-level database error (but true in case of a valid call, even if HasData will return false, i.e. no data match)

function Search(const Prepared: TSQVirtualTablePrepared): boolean; virtual; abstract;
Called to begin a search in the virtual table
- the TSQVirtualTablePrepared parameters were set by TSQVirtualTable.Prepare and will contain both WHERE and ORDER BY statements (retrieved e.g. by x_BestIndex() from a TSQLite3IndexInfo structure)
- Prepared will contain all prepared constraints and the corresponding expressions in the Where[].Value field
- should move cursor to first row of matching data
- should return false on low-level database error (but true in case of a valid call, even if HasData will return false, i.e. no data match)

property Table: TSQVirtualTable read fTable;
The associated Virtual Table class instance

TSQVirtualTableCursorIndex = class(TSQVirtualTableCursor)

A generic Virtual Table cursor associated to Current/Max index properties

function HasData: boolean; override;
Called after Search() to check if there is data to be retrieved
- will return false if reached the end of matching data, according to the fCurrent/fMax protected properties values
function Next: boolean; override;

Called to go to the next row of matching data
- will return false on low-level database error (but true in case of a valid call, even if HasData will return false, i.e. no data match)
- will check the fCurrent/fMax protected properties values

function Search(const Prepared: TSQLVirtualTablePrepared): boolean; override;

Called to begin a search in the virtual table
- this no-op version will mark EOF, i.e. fCurrent=0 and fMax=-1

TSQLVirtualTableCursorJSON = class(TSQLVirtualTableCursorIndex)

A Virtual Table cursor for reading a TSQLRestStorageInMemory content
- this is the cursor class associated to TSQLVirtualTableJSON

function Column(aColumn: integer; var aResult: TSQLVar): boolean; override;

Called to retrieve a column value of the current data row into a TSQLVar
- if aColumn=-1, will return the row ID as varInt64 into aResult
- will return false in case of an error, true on success

function Search(const Prepared: TSQLVirtualTablePrepared): boolean; override;

Called to begin a search in the virtual table
- the TSQLVirtualTablePrepared parameters were set by TSQLVirtualTable.Prepare and will contain both WHERE and ORDER BY statements (retrieved by x_BestIndex from a TSQLite3IndexInfo structure)
- Prepared will contain all prepared constraints and the corresponding expressions in the Where[].Value field
- will move cursor to first row of matching data
- will return false on low-level database error (but true in case of a valid call, even if HasData will return false, i.e. no data match)
- only handled WHERE clause is for "ID = value" - other request will return all records in ID order, and let the database engine handle it

TSQLVirtualTableJSON = class(TSQLVirtualTable)

A TSQLRestStorageInMemory-based virtual table using JSON storage
- for ORM access, you should use TSQLModel.VirtualTableRegister method to associated this virtual table module to a TSQLRecordVirtualTableAutoID class
- transactions are not handled by this module
- by default, no data is written on disk: you will need to call explicitly aServer.StaticVirtualTable[aClass].UpdateToFile for file creation or refresh
- file extension is set to '.json'
constructor Create(aModule: TSQLVirtualTableModule; const aTableName: RawUTF8; FieldCount: integer; Fields: PPUTF8CharArray); override;

Create the virtual table access instance
- the created instance will be released when the virtual table will be disconnected from the DB connection (e.g. xDisconnect method for SQLite3)
- shall raise an exception in case of invalid parameters (e.g. if the supplied module is not associated to a TSQLRestServer instance)
- aTableName will be checked against the current aModule.Server.Model to retrieve the corresponding TSQLRecordVirtualTableAutoID class and create any associated Static: TSQLRestStorage instance

function Delete(aRowID: Int64): boolean; override;
Called to delete a virtual table row
- returns true on success, false otherwise

function Drop: boolean; override;
Called when a DROP TABLE statement is executed against the virtual table
- returns true on success, false otherwise

function Insert(aRowID: Int64; var Values: TSQLVarDynArray; out insertedRowID: Int64): boolean; override;
Called to insert a virtual table row content from a TSQLVar array
- column order follows the Structure method, i.e. StoredClassRecordProps.Fields[] order
- returns true on success, false otherwise
- returns the just created row ID in insertedRowID on success
- does nothing by default, and returns false, i.e. always fails

function Prepare(var Prepared: TSQLVirtualTablePrepared): boolean; override;
Called to determine the best way to access the virtual table
- will prepare the request for TSQLVirtualTableCursor.Search()
- only prepared WHERE statement is for "ID = value"
- only prepared ORDER BY statement is for ascending IDs

function Update(oldRowID, newRowID: Int64; var Values: TSQLVarDynArray): boolean; override;
Called to update a virtual table row content from a TSQLVar array
- column order follows the Structure method, i.e. StoredClassRecordProps.Fields[] order
- returns true on success, false otherwise
- does nothing by default, and returns false, i.e. always fails

class procedure GetTableModuleProperties( var aProperties: TVirtualTableModuleProperties); override;
Returns the main specifications of the associated TSQLVirtualTableModule
- this is a read/write table, without transaction, associated to the TSQLVirtualTableCursorJSON cursor type, with 'JSON' as module name
- no particular class is supplied here, since it will depend on the associated Static instance
**TSQLVirtualTableBinary** = class(TSQLVirtualTableJSON)

- A TSQLRestStorageInMemory-based virtual table using Binary storage
- for ORM access, you should use TSQLModel.VirtualTableRegister method to associated this virtual table module to a TSQLRecordVirtualTableAutoID class
- transactions are not handled by this module
- by default, no data is written on disk: you will need to call explicitly aServer.StaticVirtualTable[aClass].UpdateToFile for file creation or refresh
- binary format is more efficient in term of speed and disk usage than the JSON format
- binary format will be set by TSQLVirtualTableJSON.CreateTableInstance
- file extension is set to '.data'

**TSQLVirtualTableLog** = class(TSQLVirtualTable)

- Implements a read-only virtual table able to access a .log file, as created by TSynLog
- to be used e.g. by a TSQLRecordLog_Log (‘Log’ will identify this ‘Log’ module)
- the .log file name will be specified by the Table Name, to which a ‘.log’ file extension will be appended before loading it from the current directory

**constructor** Create(aModule: TSQLVirtualTableModule; const aTableName: RawUTF8; FieldCount: integer; Fields: PPUTF8CharArray); override;

- Creates the TSQLVirtualTable according to the supplied parameters
- aTableName will be checked against the current aModule.Server.Model to retrieve the corresponding TSQLRecordVirtualTableAutoID class

**destructor** Destroy; override;

- Release the associated .log file mapping and all internal structures

**class procedure** GetTableModuleProperties( var aProperties: TVirtualTableModuleProperties); override;

- Returns the main specifications of the associated TSQLVirtualTableModule
- this is a read only table, with transaction, associated to the TSQLVirtualTableCursorLog cursor type, with ‘Log’ as module name, and associated to TSQLRecordLog_Log table field layout

**TSQLVirtualTableCursorLog** = class(TSQLVirtualTableCursorIndex)

- A Virtual Table cursor for reading a TSynLogFile content
- this is the cursor class associated to TSQLVirtualTableLog

**function** Column(aColumn: integer; var aResult: TSQLVar): boolean; override;

- Called to retrieve a column value of the current data row as TSQLVar

**function** Search(const Prepared: TSQLVirtualTablePrepared): boolean; override;

- Called to begin a search in the virtual table
`TSQLRecordVirtualTableForcedID = class(TSQLRecordVirtual)`

*Record associated to a Virtual Table implemented in Delphi, with ID forced at INSERT*
- will use TSQLVirtualTableModule / TSQLVirtualTable / TSQLVirtualTableCursor classes for a
generic Virtual table mechanism on the Server side
- call `Model.VirtualTableRegister()` before TSQLRestServer.Create on the Server side (not needed
for Client) to associate such a record with a particular Virtual Table module, otherwise an
exception will be raised:
```pascal
Model.VirtualTableRegister(TSQLRecordDali1,TSQLVirtualTableJSON);
```
PServiceMethod = ^TServiceMethod;

A pointer to an interface-based service provider method description
- since TInterfaceFactory instances are shared in a global list, we can safely use such pointers in our
code to refer to a particular method

PServiceMethodArgument = ^TServiceMethodArgument;

Pointer to a service provider method argument

PServiceRunningContext = ^TServiceRunningContext;

Points to the currently running service on the server side
- your code may use such a local pointer to retrieve the ServiceContext threadvar once in a method,
  since threadvar access does cost some CPU
var context: PServiceRunningContext;
begin
  context := @ServiceContext; // threadvar access once
  ...

PSQLRecordPropertiesMapping = ^TSQLRecordPropertiesMapping;

Pointer to external database properties for ORM
- is used e.g. to allow a “fluent” interface for MapField() method

PSQLRestClientCallbackItem = ^TSQLRestClientCallbackItem;

Points to information about registered interface callbacks

PSQLRestServerKind = ^TSQLRestServerKind;

Pointer to the kind of (static) database server implementation

PSQLRestServerMethod = ^TSQLRestServerMethod;

Pointer to a description of a method-based service

PSQLRestURIParams = ^TSQLRestURIParams;

Used to map set of parameters for a Client or Server method call

PSQLTableRowVariantData = ^TSQLTableRowVariantData;

Pointer to the memory structure used for TSQLTableRowVariant storage

PTypeInfo = ^TTypeInfo;

Not defined on older Delphi revisions

TAuthSessionClass = class of TAuthSession;

Class-reference type (metaclass) used to define overridden session instances
- since all sessions data remain in memory, ensure they are not taking too much resource (memory
  or process time)
- if you plan to use session persistence, ensure you override the TAuthSession.SaveTo/CreateFrom
  methods in the inherited class

TCallingConvention = ( ccRegister, ccCdecl, ccPascal, ccStdCall, ccSafeCall );

The available methods calling conventions
- this is by design only relevant to the x86 model
- Win64 has one unique calling convention

TClassInstanceItemCreate = ( cicUnknown, cicTSQLRecord, cicTObjectList, cicTPersistentWithCustomCreate, cicTSynPersistent, cicTInterfacedCollection, cicTInterfacedObjectWithCustomCreate, cicTCollection, cicTCollectionItem, cicTComponent, cic TObject );
**The class kind as handled by TClassInstance object**

<table>
<thead>
<tr>
<th>TCreateTime = type</th>
<th>TTimeLog;</th>
</tr>
</thead>
<tbody>
<tr>
<td>An Int64-encoded date and time of the record creation</td>
<td></td>
</tr>
<tr>
<td>- can be used as published property field in TSQLRecord for sftCreateTime: if any such property is defined in the table, it will be auto-filled with the server timestamp corresponding to the record creation</td>
<td></td>
</tr>
<tr>
<td>- use internally for computation an abstract &quot;year&quot; of 16 months of 32 days of 32 hours of 64 minutes of 64 seconds - faster than TDateTime</td>
<td></td>
</tr>
<tr>
<td>- use TimeLogFromDateTime/TimeLogToDateTime/TimeLogNow/Iso8601ToTimeLog functions, or type-cast the value with a TTimeLogBits memory structure for direct access to its bit-oriented content (or via PTimeLogBits pointer)</td>
<td></td>
</tr>
<tr>
<td>- could be defined as value in a TSQLRecord property as such:</td>
<td></td>
</tr>
<tr>
<td>property CreatedAt: TModTime read fCreatedAt write fcreatedAt;</td>
<td></td>
</tr>
</tbody>
</table>

| TFindWhereEqualEvent = procedure(aDest: pointer; aRec: TSQLRecord; aIndex: integer) of object; |
| Event prototype called by TSQLRestStorageInMemory.FindWhereEqual() or TSQLRestStorageInMemory.ForEach() methods |
| - aDest is an opaque pointer, as supplied to FindWhereEqual(), which may point e.g. to a result list, or a shared variable to apply the process |
| - aRec will point to the corresponding item |
| - aIndex will identify the item index in the internal list |

| TID = type | Int64; |
| This is the type to be used for our ORM primary key, i.e. TSQLRecord.ID |
| - it maps the SQLite3 RowID definition |
| - when converted to plain TSQLRecord published properties, you may loose some information under Win32 when stored as a 32-bit pointer |
| - could be defined as value in a TSQLRecord property as such: |
| property AnotherRecord: TID read fAnotherRecord write fAnotherRecord; |

| TIDDynArray = array of TID; |
| Used to store a dynamic array of ORM primary keys, i.e. TSQLRecord.ID |

| TInjectableObjectClass = class of TInjectableObject; |
| Class-reference type (metaclass) of a TInjectableObject type |

| TInjectableObjectRestClass = class of TInjectableObjectRest; |
| Class-reference type (metaclass) of a TInjectableObjectRest type |

| TInterfacedCollectionClass = class of TInterfacedCollection; |
| Class-reference type (metaclass) of a TInterfacedCollection kind |

| TInterfacedObjectFromFactoryOption = (ifoJsonAsExtended, ifoDontStoreVoidJSON); |
| How TInterfacedObjectFromFactory will perform its execution |
| - by default, fInvoke() will receive standard JSON content, unless ifoJsonAsExtended is set, and extended JSON is used |
| - ifoDontStoreVoidJSON will ensure objects and records won't include default void fields in JSON serialization |

| TInterfacedObjectFromFactoryOptions = set of TInterfacedObjectFromFactoryOption; |
Defines how TInterfacedObjectFromFactory will perform its execution

TInterfacedObjectObjArray = array of TInterfacedObject;

Used to store a list of TInterfacedObject instances

TInterfaceFactoryMethodBits = set of 0..MAX_METHOD_COUNT-1;

May be used to store the Methods[] indexes of a TInterfaceFactory
- current implementation handles up to 128 methods, a limit above which "Interface Segregation" principles is obviously broken

TInterfaceFactoryObjArray = array of TInterfaceFactory;

A dynamic array of TInterfaceFactory instances

TInterfaceMockSpyCheck = (chkName, chkNameParams, chkNameParamsResults);

How TInterfaceMockSpy.Verify() shall generate the calls trace

TInterfaceResolverObjArray = array of TInterfaceResolver;

Used to store a list of TInterfaceResolver instances

TInterfaceStubLogDynArray = array of TInterfaceStubLog;

Used to keep track of all stubbed methods calls

TInterfaceStubLogLayout = (wName, wParams, wResults);

Every potential part of TInterfaceStubLog.AddAsText() log entry

TInterfaceStubLogLayouts = set of TInterfaceStubLogLayout;

Set the output layout of TInterfaceStubLog.AddAsText() log entry

TInterfaceStubObjArray = array of TInterfaceStub;

Used to store a list of TInterfaceStub instances

TInterfaceStubOption = (imoLogMethodCallsAndResults, imoFakeInstanceOfReleaseTInterfaceStub, imoRaiseExceptionIfNoRuleDefined, imoReturnErrorIfNoRuleDefined, imoMockFailsWillPassTestCase);

Diverse options available to TInterfaceStub
- by default, method execution stack is not recorded - include imoLogMethodCallsAndResults in the options to track all method calls and the returned values; note that ExpectsTrace() method will set it
- by default, TInterfaceStub will be released when the stubed/mockied interface is released - include imoFakeInstanceOfReleaseTInterfaceStub in the options to force manual memory handling of TInterfaceStubs
- by default, all interfaces will return some default values, unless imoRaiseExceptionIfNoRuleDefined or imoReturnErrorIfNoRuleDefined is included in the options
- by default, any TInterfaceMock.Fails() rule execution will notify the TSynTestCase, unless imoMockFailsWillPassTestCase which will let test pass

TInterfaceStubOption = set of TInterfaceStubOption;

Set of options available to TInterfaceStub

TInterfaceStubRuleKind = (isUndefined, isExecutesJSON, isExecutesVariant, isRaises, isReturns, isFails);

Diverse types of stubbing / mocking rules
- isUndefined is the first, since it will be a ExpectsCount() weak rule which may be overwritten by the other real run-time rules

TJSONObjectDecoderFieldType = (ftaNumber, ftaBoolean, ftaString, ftaDate, ftaNull, ftaBlob, ftaObject, ftaArray);
Define how TJSONObjectDecoder.FieldTypeApproximation[] is identified

TJSONObjectDecoderParams = ( pInlined, pQuoted, pNonQuoted );

Define how TJSONObjectDecoder.Decode() will handle JSON string values

TJSONObjectDecoderCustomReader = function(const aValue: TObject; aFrom: PUTF8Char; var aValid: Boolean; aOptions: TJSONToObjectOptions): PUTF8Char of object;

Method prototype to be used for custom un-serialization of a class
- to be used with TJSONObjectDecoder.RegisterCustomSerializer() method
- note that the read JSON content is not required to start with ',', as a normal JSON object (you may e.g. read a JSON string for some class) - as a consequence, custom code could explicitly start with "if aFrom^='{'..."
- implementation code shall follow function JSONToObject() patterns, i.e. calling low-level GetJSONField() function to decode the JSON content
- implementation code shall follow the same exact format for the associated TJSONSerializerCustomReader callback

TJSONObjectDecoderCustomWriter = procedure(const aSerializer: TJSONSerializer; aValue: TObject; aOptions: TTextWriterWriteObjectOptions) of object;

Method prototype to be used for custom serialization of a class
- to be used with TJSONObjectDecoder.RegisterCustomSerializer() method
- note that the generated JSON content is not required to start with ',', as a normal JSON object (you may e.g. write a JSON string for some class) - as a consequence, custom code could explicitly start with Add('')
- implementation code shall follow function TJSONSerializer.WriteObject() patterns, i.e. aSerializer.Add/AddInstanceName/AddJSONEscapeString...
- implementation code shall follow the same exact format for the associated TJSONSerializerCustomWriter callback

TJSONSerializerSQLRecordOption = ( jwoAsJsonNotAsString, jwoID_str );

Several options to customize how TSQLRecord will be serialized
- e.g. if properties storing JSON should be serialized as an object, and not escaped as a string (which is the default, matching ORM column storage)
- if an additional "ID_str":"12345" field should be added to the standard "ID":12345 field, which may exceed 53-bit integer precision of JavaSript

TJSONSerializerSQLRecordOptions = set of TJSONSerializerSQLRecordOption;

Options to customize how TSQLRecord will be written by TJSONSerializer

TJSONObjectToObjectOption = ( j2oIgnoreUnknownProperty, j2oIgnoreStringType,
j2oIgnoreUnknownEnum, j2oHandleCustomVariants, j2oHandleCustomVariantsWithinString,
j2oSetterExpectsToFreeTempInstance, j2oSetterNoCreate, j2oAllowInt64Hex );

Available options for JSONToObject() parsing process
- by default, function will fail if a JSON field name is not part of the object published properties, unless j2oIgnoreUnknownProperty is defined - this option will also ignore read-only properties (i.e. with only a getter)
- by default, function will check that the supplied JSON value will be a JSON string when the property is a string, unless j2oIgnoreStringType is defined and JSON numbers are accepted and stored as text
- by default any unexpected value for enumerations will be marked as invalid, unless j2oIgnoreUnknownEnum is defined, so that in such case the ordinal 0 value is left, and loading continues
- by default, only simple kind of variant types (string/numbers) are handled: set j2oHandleCustomVariants if you want to handle any custom - in this case, it will handle direct JSON
[array] of (object); but if you also define j2oHandleCustomVariantsWithinString, it will also try to un-escape a JSON string first, i.e. handle "[array]" or "{object}" content (may be used e.g. when JSON has been retrieved from a database TEXT column)
- by default, a temporary instance will be created if a published field has a setter, and the instance is expected to be released later by the owner class: set j2oSetterExpectsToFreeTempInstance to let JSONToObject (and TPropInfo.ClassFromJSON) release it when the setter returns, and j2oSetterNoCreate to avoid the published field instance creation
- set j2oAllowInt64Hex to let Int64/QWord fields accept hexadecimal string (as generated e.g. via the w𝚘n𝚝64AsHex option)

TJSONToObjectOptions = set of TJSONToObjectOption;
Set of options for JSONToObject() parsing process

TModTime = type TTimeLog;
An Int64-encoded date and time of the latest update of a record
- can be used as published property field in TSQLRecord for sftModTime: if any such property is defined in the table, it will be auto-filled with the server timestamp corresponding to the latest record update
- use internally for computation an abstract "year" of 16 months of 32 days of 64 minutes of 64 seconds - faster than TDateTime
- use TimeLogFromDateTime/TimeLogToDateTime/TimeLogNow/Iso8601ToTimeLog functions, or type-cast the value with a TTimeLogBits memory structure for direct access to its bit-oriented content (or via PTimeLogBits pointer)
- could be defined as value in a TSQLRecord property as such:

property LastModif: TModTime read fLastModif write fLastModif;

TNotifyAfterURI = procedure(Ctxt: TSQLRestServerURIContext) of object;
Callback raised after TSQLRestServer.URI execution

TNotifyAuthenticationFailed = procedure(Sender: TSQLRestServer; Reason: TNotifyAuthenticationFailedReason; Session: TAuthService; Ctxt: TSQLRestServerURIContext) of object;
Callback raised in case of authentication failure
- as used by TSQLRestServerURIContext.AuthenticationFailed event

TNotifyAuthenticationFailedReason = ( afInvalidSignature, afRemoteServiceExecutionNotAllowed, afUnknownUser, afInvalidPassword, afSessionAlreadyStartedForThisUser, afSessionCreationAborted, afSecureConnectionRequired, afJWTRequired );

Used to identify the authentication failure reason
- as transmitted e.g. by TSQLRestServerURIContext.AuthenticationFailed or TSQLRestServer.OnAuthenticationFailed

TNotifyBeforeURI = function(Ctxt: TSQLRestServerURIContext): boolean of object;
Callback raised before TSQLRestServer.URI execution
- should return TRUE to execute the command, FALSE to cancel it

TNotifyErrorURI = function(Ctxt: TSQLRestServerURIContext; E: Exception): boolean of object;
Callback raised if TSQLRestServer.URI execution failed
- should return TRUE to execute Ctx.Error(E,...), FALSE if returned content has already been set as expected by the client
TNotifyFieldSQLEvent = function(Sender: TSQLRestServer; Event: TSQLEvent; aTable: TSQLRecordClass; const aID: TID; const aAffectedFields: TSQLFieldBit): boolean of object;

Used to define how to trigger Events on record field update
- see TSQLRestServer.OnBlobUpdateEvent property and InternalUpdateEvent() method
- returns true on success, false if an error occured (but action must continue)
- to be used only server-side, not to synchronize some clients: the framework is designed around a stateless RESTful architecture (like HTTP/1.1), in which clients ask the server for refresh (see TSQLRestClientURI.UpdateFromServer)

TNotifyRestBody = procedure(Sender: TSQLRest; var Body, Head, URL: RawUTF8) of object;

Callback event signature before/after a Client or Server method call
- to allow low-level interception of the request bodies e.g. for low-level logging/audit, or on-the-fly encryption and/or signature of the content
- used by TSQLRest.OnDecryptBody and TSQLRest.OnEncryptBody - so the very same callbacks may be used on both client and server sides
- for server-only process (e.g. to check for authorization), see rather TSQLRestServer.OnBeforeURI and TSQLRestServer.OnAfterURI events
- used e.g. by TSQLRest.SetCustomEncryption method

TNotifySQLEvent = function(Sender: TSQLRestServer; Event: TSQLEvent; aTable: TSQLRecordClass; const aID: TID; const aAffectedFields: TSQLFieldBits): boolean of object;

Used to define how to trigger Events on record update
- see TSQLRestServer.OnUpdateEvent property and InternalUpdateEvent() method
- returns true on success, false if an error occured (but action must continue)
- to be used only server-side, not to synchronize some clients: the framework is designed around a stateless RESTful architecture (like HTTP/1.1), in which clients ask the server for refresh (see TSQLRestClientURI.UpdateFromServer)

TNotifySQLSession = function(Sender: TSQLRestServer; Session: TAuthSession; Ctxt: TSQLRestServerURIContext): boolean of object;

Session-related callbacks triggered by TSQLRestServer
- for OnSessionCreate, returning TRUE will abort the session creation - and you can set Ctxt.Call^.OutStatus to a corresponding error code

TOnAsynchRedirectResult = procedure(const aMethod: TServiceMethod; const aInstance: IInvokable; const aParams, aResult: RawUTF8) of object;

Optionally called after TSQLRest.AsynchRedirect background execution
- to retrieve any output result value, as JSON-encoded content
- as used in TSQLRestBackgroundTimer.AsynchBackgroundExecute protected method

TOnAuthenticationUserRetrieve = function(Sender: TSQLRestServerAuthentication; Ctxt: TSQLRestServerURIContext; aUserID: TID; const aUserName: RawUTF8): TSQLAuthUser of object;

Callback allowing to customize the retrieval of an authenticated user
- as defined in TSQLRestServer.OnAuthenticationUserRetrieve
- and executed by TSQLRestServerAuthentication.GetUser
- on call, either aUserID will be <> 0, or aUserName is to be used
- if the function returns nil, default Server.SQLAuthUserClass.Create() methods won't be called, and the user will be reported as not found

TOnAuthenticationFailed = function(Retry: integer; var aUserName, aPassword: string; out aPasswordHashed: boolean): boolean of object;
**Used by TSQLRestClientURI.URI() to let the client ask for an User name and password, in order to retry authentication to the server**

- should return TRUE if aUserName and aPassword both contain some entered values to be sent for remote secure authentication
- should return FALSE if the user pressed cancel or the number of Retry reached a defined limit
- here input/output parameters are defined as plain string, to match the type expected by the client's User Interface, via VCL properties, or e.g. from TLoginForm as defined in mORMotUILogin.pas unit

```
TOnBatchWrite = procedure(Sender: TSQLRestBatch; Event: TSQLOccasion; Table: TSQLRecordClass; const ID: TID; Value: TSQLRecord; const ValueFields: TSQLFieldBits)
  of object;
```

**Event signature triggered by TSQLRestBatch.OnWrite**

- also used by TSQLRestServer.RecordVersionSynchronizeSlave*() methods

```
TOnCallbackReleased = procedure(Sender: TServiceContainer; Instance: TInterfacedObject; Callback: pointer) of object;
```

**Event signature triggered when a callback instance is released**

- used by TServiceContainerServer.OnCallbackReleasedOnClientSide and TServiceContainerServer.OnCallbackReleasedOnServerSide event properties
- the supplied Instance will be a TInterfacedObjectFakeServer, and the Callback will be a pointer to the corresponding interface value
- assigned implementation should be as fast a possible, since this event will be executed in a global lock for all server-side callbacks

```
TOnClientFailed = procedure(Sender: TSQLRestClientURI; E: Exception; Call: PSQLRestURIParams)
  of object;
```

**Called by TSQLRestClientURI.URI() when an error occurred**

- so that you may have a single entry point for all client-side issues
- information will be available in Sender's LastErrorCode and LastErrorMessage properties
- if the error comes from an Exception, it will be supplied as parameter
- the REST context (if any) will be supplied within the Call parameter, and in this case Call^.OutStatus=HTTP_NOTIMPLEMENTED indicates a broken connection

```
TOnFakeInstanceDestroy = procedure(aClientDrivenID: cardinal)
  of object;
```

**Event called when destroying a TInterfaceFactory's fake instance**

- this method will be run when the fake class instance is destroyed (e.g. if aInstanceCreation is sicClientDriven, to notify the server than the client life time just finished)

```
TOnFakeInstanceInvoke = function(const aMethod: TServiceMethod; const aParams: RawUTF8; aResult, aErrorMsg: PRawUTF8; aClientDrivenID: PCardinal; aServiceCustomAnswer: PServiceCustomAnswer): boolean
  of object;
```

**Event used by TInterfaceFactory to run a method from a fake instance**

- aMethod will specify which method is to be executed
- aParams will contain the input parameters, encoded as a JSON array, without the [ ] characters (e.g. '1","arg2",3')
- shall return TRUE on success, or FALSE in case of failure, with a corresponding explanation in aErrorMsg
- method results shall be serialized as JSON in aResult; if aServiceCustomAnswer is not nil, the result shall use this record to set HTTP custom content and headers, and ignore aResult content
- aClientDrivenID can be set optionally to specify e.g. an URI-level session

```
TOnInterfaceStubExecuteJSON = procedure(Ctx: TOnInterfaceStubExecuteParamsJSON)
```
**object**

*Event called by the TInterfaceStub.Executes() fluent method for JSON process*
- by default Ctxt.Result shall contain the default JSON array result for this method - use Ctxt.Named[]
  default properties, e.g. as
  ```pascal
  P := pointer(Ctxt.Params);
  Ctxt.Returns([GetNextItemDouble(P)-GetNextItemDouble(P)]);
  ```
- you can call Ctxt.Error() to notify the caller for an execution error

**TOnInterfaceStubExecuteVariant = procedure(Ctxt: TOnInterfaceStubExecuteParamsVariant) of object;**

*Event called by the TInterfaceStub.Executes() fluent method for variant process*
- by default Ctxt.Result shall contain the default JSON array result for this method - use Ctxt.Named[]
  default properties, e.g. as
  ```pascal
  Ctxt['result'] := Ctxt['n1']-Ctxt['n2'];
  ```
  or with Input[] / Output[] properties:
  ```pascal
  with Ctxt do Output[0] := Input[0]-Input[1];
  ```
- you can call Ctxt.Error() to notify the caller for an execution error

**TOnInternalInfo = procedure(Sender: TSQLRestServer; var info: TDocVariantData) of object;**

*Callback allowing to customize the information returned by root/timestamp/info*

**TOnRecordUpdate = procedure(Value: TSQLRecord) of object;**

*Used by TSQLRestClientURI.Update() to let the client perform the record update (refresh associated report e.g.)*

**TOnRestClientNotify = procedure(Sender: TSQLRestClientURI) of object;**

*Signature e.g. of the TSQLRestClientURI.OnSetUser event handler*

**TOnServiceCanExecute = function(Ctxt: TSQLRestServerURIContext; const Method: TServiceMethod): boolean of object;**

*Callback called before any interface-method service execution to allow its execution*
- see Ctxt.Service, Ctxt.ServiceMethodIndex and Ctxt.ServiceParameters to identify the executed method context
- Method parameter will help identify easily the corresponding method, and will contain in fact PServiceMethod(Ctxt.ServiceMethod)^
- should return TRUE if the method can be executed
- should return FALSE if the method should not be executed, and set the corresponding error to the supplied context e.g.
  ```pascal
  Ctxt.Error('Unauthorized method',HTTP_NOTALLOWED);
  ```
- i.e. called by TSQLRestServerURIContext.InternalExecuteSOAByInterface

**TOnService.CreateInstance = procedure( Sender: TServiceFactoryServer; Instance: TInterfacedObject) of object;**

*Event signature used by TSQLRestServer.OnService.CreateInstance*
- as called by TServiceFactoryServer.CreateInstance
- the actual Instance class can be quickly retrieved from Sender.ImplementationClass

**TOnServiceFactoryServerOne = function(Sender: TServiceFactoryServer; var Instance: TServiceFactoryServerInstance; var Opaque): integer of object;**

*Callbacked used by TServiceFactoryServer.RunOnAllInstances method*

**TOnSQLPropInfoRecord2Data = procedure(Text: PUTF8Char; var Data: RawByteString);**

*Optional event handler used by TSQLPropInfoRecord to handle textual storage*
- by default, TSQLPropInfoRecord content will be stored as sftBlobCustom; specify such a callback event to allow storage as UTF-8 textual field and use a sftUTF8Custom kind of column
  - event implementation shall convert Text into Data binary value

```pascal
TOnSQLPropInfoRecord2Text = procedure(Data: pointer; DataLen: integer; var Text: RawUTF8);
```

*Optional event handler used by TSQLPropInfoRecord to handle textual storage*
- by default, TSQLPropInfoRecord content will be stored as sftBlobCustom; specify such a callback event to allow storage as UTF-8 textual field and use a sftUTF8Custom kind of column
- event implementation shall convert data/datalen binary value into Text

```pascal
TOnSQLTableGetValue = function(Sender: TSQLTable; Row, Field: integer; HumanFriendly: boolean): RawJSON of object;
```

-Allow on-the-fly translation of a TSQLTable grid value
- should return valid JSON value of the given cell (i.e. quoted strings, or valid JSON object/array) unless HumanFriendly is defined
- e.g. TSQLTable.OnExportValue property will customize TSQLTable's GetJSONValues, GetHtmlTable, and GetCSVValues methods returned content

```pascal
TOnTableUpdate = procedure(aTable: TSQLTableJSON; State: TOnTableUpdateState) of object;
```

*Used by TSQLRestClientURI.UpdateFromServer() to let the client perform the rows update (for Marked[] e.g.)*

```pascal
TOnTableUpdateState = ( tusPrepare, tusChanged, tusNoChange );
```

*Possible call parameters for TOnTableUpdate Event*

```pascal
TParamFlag = ( pfVar, pfConst, pfArray, pfAddress, pfReference, pfOut, pfResult );
```

*The available kind of method parameters*

```pascal
TPropInfoCall = ( picNone, picField, picMethod, picIndexed );
```

*How a RTTI property definition access its value*
- as returned by TPropInfo.Getter/Setter methods

```pascal
TRawUTF8ObjectCacheClass = class of TRawUTF8ObjectCache;
```

*Class-reference type (metaclass) of a TRawUTF8ObjectCache*
- used e.g. by TRawUTF8ObjectCacheClass.Create to generate the expected cache instances

```pascal
TRecordReference = type Int64;
```

*A reference to another record in any table in the database Model*
- stored as a 64-bit signed integer (just like the TID type)
- type cast any value of TRecordReference with the RecordRef object below for easy access to its content
- use TSQLRest.Retrieve(Reference) to get a record value
- don't change associated TSQLModel tables order, since TRecordReference depends on it to store the Table type in its highest bits
- when the pointed record will be deleted, this property will be set to 0 by TSQLRestServer.AfterDeleteForceCoherency()
- could be defined as value in a TSQLRecord property as such:

```pascal
property AnotherRecord: TRecordReference read fAnotherRecord write fAnotherRecord;
```
TRecordReferenceToBeDeleted = type TRecordReference;

A reference to another record in any table in the database Model
- stored as a 64-bit signed integer (just like the TID type)
- type cast any value of TRecordReference with the RecordRef object below for easy access to its content
- use TSQRLRest.Restore(Reference) to get a record value
- don’t change associated TSQLModel tables order, since TRecordReference depends on it to store the Table type in its highest bits
- when the pointed record will be deleted, any record containing a matching property will be deleted by TSQRLRestServer.AfterDeleteForceCoherency()
- could be defined as value in a TSQLRecord property as such:

property AnotherRecord: TRecordReferenceToBeDeleted
  read fAnotherRecord write fAnotherRecord;

TRecordVersion = type Int64;

A monotonic version number, used to track changes on a table
- add such a published field to any TSQLRecord will allow tracking of record modifications - note that only one field of this type should be defined for a given record
- note that this published field is NOT part of the record "simple fields": by default, the version won't be retrieved from the DB, nor will be sent from a client - the Engine*() CRUD method will take care of computing the monotonic version number, just before storage to the persistence engine
- such a field will use a separated TSQRLRecordTableDeletion table to track the deleted items
- could be defined as value in a TSQLRecord property as such:

property TrackedVersion: TRecordVersion read fVersion write fVersion;

TSERVICECallbackOptions = set of ( coRaiseExceptionIfReleasedByClient);

How TServiceContainerServer will handle SOA callbacks
- by default, a callback released on the client side will log a warning and continue the execution (relying e.g. on a CallbackReleased() method to unsubscribe the event), but coRaiseExceptionIfReleasedByClient can be defined to raise an EInterfaceFactoryException in this case

TSERVICEContainerInterfaceMethodBits = set of 0..255;

Used in TServiceContainer to identify fListInterfaceMethod[] entries

TSERVICEContainerInterfaceMethods = array of TServiceContainerInterfaceMethod;

Used to store all methods in a global list of interface-based services

TSERVICEContainerInterfaces = array of TServiceContainerInterface;

Used to store all s in a global list of interface-based services

TSERVICEFactoryClientClass = class of TServiceFactoryClient;

Class-reference type (metaclass) of a TServiceFactoryClient kind

TSERVICEFactoryServerInstanceDynArray = array of TServiceFactoryServerInstance;

Server-side service provider uses this to store its internal instances
- used by TServiceFactoryServer in sicClientDriven, sicPerSession, sicPerUser or sicPerGroup mode

TSERVICEInstanceImplementation = ( sicSingle, sicShared, sicClientDriven,
  sicPerSession, sicPerUser, sicPerGroup, sicPerThread );

The possible Server-side instance implementation patterns for interface-based services
- each interface-based service will be implemented by a corresponding class instance on the server:
  this parameter is used to define how class instances are created and managed
- on the Client-side, each instance will be handled depending on the server side implementation (i.e.
  with sicClientDriven behavior if necessary)
- sicSingle: one object instance is created per call - this is the most expensive way of implementing
  the service, but is safe for simple workflows (like a one-type call); this is the default setting for
  TSQLRestServer.ServiceRegister method
- sicClientShared: one object instance is used for all incoming calls and is not recycled subsequent to the
  calls - the implementation should be thread-safe on the server side
- sicClientDriven: one object instance will be created in synchronization with the client-side lifetime
  of the corresponding interface: when the interface will be released on client, it will be released on
  the server side - a numerical identifier will be transmitted for all JSON requests
- sicPerSession, sicPerUser and sicPerGroup modes will maintain one object instance per running
  session / user / group (only working if RESTful authentication is enabled) - since it may be shared
  among users or groups, the sicPerUser and sicPerGroup implementation should be thread-safe
- sicPerThread will maintain one object instance per calling thread - it may be useful instead of
  sicShared mode if the service process expects some per-heavy thread initialization, for instance

```pascal
TServInstanceImplementations = set of TServiceInstanceImplementation;

Set of Server-side instance implementation patterns for interface-based services
```

```pascal
TServMethodArgumentDynArray = array of TServiceMethodArgument;

Describe a service provider method arguments
```

```pascal
TServMethodDynArray = array of TServiceMethod;

Describe all methods of an interface-based service provider
```

```pascal
TServMethodExecuteCallback = procedure(var Par: PUTF8Char; ParamInterfaceInfo: PTypeInfo; out Obj) of object;

Callback called by TServiceMethodExecute to process an interface callback parameter
- implementation should set the Obj local variable to an instance of a fake class implementing the
  aParamInfo interface
```

```pascal
TServMethodExecuteEvent = procedure(Sender: TServiceMethodExecute; Step: TServiceMethodExecuteEventStep) of object;

The TServiceMethodExecute.OnExecute signature
```

```pascal
TServMethodExecuteEventStep = ( smsUndefined, smsBefore, smsAfter, smsError );

The current step of a TServiceMethodExecute.OnExecute call
```

```pascal
TServMethodOption = ( optExecLockedPerInterface, optExecInPerInterfaceThread, optFreeInPerInterfaceThread, optExecInMainThread, optFreeInMainThread, optVariantCopiedByReference, optInterceptInputOutput, optNoLogInput, optNoLogOutput, optErrorOnMissingParam, optForceStandardJSON, optDontStoreVoidJSON, optIgnoreException );

Possible service provider method options, e.g. about logging or execution
- see TServiceMethodOptions for a description of each available option
```

```pascal
TServMethodOptions = set of TServiceMethodOption;

Set of per-method execution options for an interface-based service provider
- by default, method executions are concurrent, for better server responsiveness; if you set
  optExecLockedPerInterface, all methods of a given interface will be executed with a critical section
  - optExecInMainThread will force the method to be called within a RunningThread.Synchronize() call
  - it can be used e.g. if your implementation rely heavily on COM servers - by default, service methods
```
are called within the thread which received them, on multi-thread server instances (e.g. TSQLite3HttpServer or TSQLRestServerNamedPipeResponse), for better response time and CPU use (this is the technical reason why service implementation methods have to handle multi-threading safety carefully, e.g. by using TRTCriticalSection mutex on purpose) - optFreeInMainThread will force the _Release/Destroy method to be run in the main thread: setting this option for any method will affect the whole service class - is not set by default, for performance reasons - optExecInPerInterfaceThread and optFreeInPerInterfaceThread will allow creation of a per-interface dedicated thread - if optInterceptorInputOutput is set, TServiceFactoryServer.AddInterceptor() events will have their Sender/Input/output values defined - if optNoLogInput/optNoLogOutput is set, TSynLog and ServiceLog() database won't log any parameter values at input/output - this may be useful for regulatory/safety purposes, e.g. to ensure that no sensitive information (like a credit card number or a password), is logged during process - consider using TInterfaceFactory.RegisterUnsafeSPIType() instead if you prefer a more tuned filtering, for specific high-level types - when parameters are transmitted as JSON object, any missing parameter will be replaced by their default value, unless optErrorOnMissingParam is defined to reject the call - by default, it will check for the client user agent, and use extended JSON if none is found (e.g. from WebSockets), or if it contains 'mORMot': you can set optForceStandardJSON to ensure standard JSON is always returned - optDontStoreVoidJSON will reduce the JSON object verbosity by not writing void (e.g. 0 or '') properties when serializing objects and records - any exceptions will be propagated during execution, unless optIgnoreException is set and the exception is trapped (not to be used unless you know what you are doing)

TServiceMethodOptionsAction = ( moaReplace, moaInclude, moaExclude );

How TServiceFactoryServer.SetOptions() will set the options value

TServiceMethodParamsDocVariantKind = ( pdvArray, pdvObject, pdvObjectFixed );

How TServiceMethod.TServiceMethod method will return the generated document - will return either a dvObject or dvArray TDocVariantData, depending on the expected returned document layout - returned content could be "normalized" (for any set or enumerate) if Kind is pdvObjectFixed

TServiceMethodValueAsm = set of ( vIsString, vPassedByReference, vIsObjArray, vIsSPI, vIsQword, vIsDynArrayString, vIsDateTimeMS );

Set of low-level processing options at assembly level - vIsString is included for smvRawUTF8, smvString, smvRawByteString and smvWideString kind of parameter (smvRecord has it to false, even if they are Base-64 encoded within the JSON content, and also smvVariant/smvRawJSON) - vPassedByReference is included if the parameter is passed as reference (i.e. defined as var/out, or is a record or a reference-counted type result) - vIsObjArray is set if the dynamic array is a T*ObjArray, so should be cleared with ObjArrClear() and not TDynArray.Clear - vIsSPI indicates that the value contains some Sensitive Personal Information (e.g. a bank card number or a plain password), which type has been previously registered via TInterfaceFactory.RegisterUnsafeSPIType so that low-level logging won't include such values - vIsQword is set for ValueType=smvInt64 over a QWord unsigned 64-bit value - vIsDynArrayString is set for ValueType=smvDynArray of string values - vIsDateTimeMS is set for ValueType=smvDateTime and TDateTimeMS value
TServiceMethodValueDirection = ( smdConst, smdVar, smdOut, smdResult );

Handled kind of parameters direction for an interface-based service method
- IN, IN/OUT, OUT directions can be applied to arguments, and will be available through our JSON-serialized remote access: smdVar and smdOut kind of parameters will be returned within the "result": JSON array
- smdResult is used for a function method, to handle the returned value

TServiceMethodValueDirections = set of TServiceMethodValueDirection;

Set of parameters direction for an interface-based service method

TServiceMethodValueType = ( smvNone, smvSelf, smvBoolean, smvEnum, smvSet, smvInteger, smvCardinal, smvInt64, smvDouble, smvDateTime, smvCurrency, smvRawUTF8, smvString, smvRawByteString, smvWideString, smvBinary, smvRecord, smvVariant, smvObject, smvRawJSON, smvDynArray, smvInterface );

Handled kind of parameters for an interface-based service provider method
- we do not handle all kind of Delphi variables, but provide some enhanced types handled by JSONToObject/ObjectToJSON functions (smvObject) or TDynArray.LoadFromJSON / TTextWriter.AddDynArrayJSON methods (smvDynArray)
- records will be serialized as Base64 string, with our RecordSave/RecordLoad low-level format by default, or as true JSON objects since Delphi 2010 or after registration via a TTextWriter.RegisterCustomJSONSerializer call
- smvRawJSON will transmit the raw JSON content, without serialization

TServiceMethodValueTypes = set of TServiceMethodValueType;

Set of parameters for an interface-based service provider method

TServiceMethodValueVar = ( smvvNone, smvvSelf, smvv64, smvvRawUTF8, smvvString, smvvWideString, smvvRecord, smvvObject, smvvDynArray, smvvInterface );

Handled kind of parameters internal variables for an interface-based method
- reference-counted variables will have their own storage
- all non referenced-counted variables are stored within some 64-bit content
- smvVariant kind of parameter will be handled as a special smvvRecord

TServicesPublishedInterfacesDynArray = array of TServicesPublishedInterfaces;

Store a list of published Services supported by a TSQLRestServer instance

TSessionUserID = type TID;

The Int64/TID of the TSQLAuthUser currently logged
- can be used as published property field in TSQLRecord for sftSessionUserID: if any such property is defined in the table, it will be auto-filled with the current TSQLAuthUser.ID value at update, or 0 if no session is running
- could be defined as value in a TSQLRecord property as such:

```delphi
property User: TSessionUserID read fUser write fUser;
```

TSQLAction = ( actNoAction, actMark, actUnmarkAll, actmarkAllEntries, actmarkToday, actmarkThisWeek, actmarkThisMonth, actmarkYesterday, actmarkLastWeek, actmarkLastMonth, actmarkOlderThanOneDay, actmarkOlderThanOneWeek, actmarkOlderThanOneMonth, actmarkOlderThanSixMonths, actmarkOlderThanOneYear, actmarkInverse );

Standard actions for User Interface generation
- actNoAction for not defined action
- actMark (standard action) to Mark rows, i.e. display sub-menu with actmarkAllEntries..actmarkOlderThanOneYear items
- actUnmarkAll (standard action) to UnMark all rows
- actmarkAllEntries to Mark all rows
- actmarkToday to Mark rows for today
- actmarkThisWeek to Mark rows for this Week
- actmarkThisMonth to Mark rows for this month
- actmarkYesterday to Mark rows for today
- actmarkLastWeek to Mark rows for Last Week
- actmarkLastMonth to Mark rows for Last month
- actmarkOlderThanOneDay to Mark rows After one day
- actmarkOlderThanOneWeek to Mark rows older than one week
- actmarkOlderThanOneMonth to Mark rows older than one month
- actmarkOlderThanSixMonths to Mark rows older than one half year
- actmarkOlderThanOneYear to Mark rows older than one year
- actmarkInverse to Inverse Mark values (ON->OFF, OFF->ON)

TSQLActions = set of TSQLAction;

Set of standard actions for User Interface generation

TSQLAllowRemoteExecute = set of ( reSQL, reService, reUrlEncodedSQL, reUrlEncodedDelete, reOneSessionPerUser, reSQLSelectWithoutTable, reUserCanChangeOwnPassword );

A set of potential actions to be executed from the server
- reSQL will indicate the right to execute any POST SQL statement (not only SELECT statements)
- reSQLSelectWithoutTable will allow executing a SELECT statement with arbitrary content via GET/LOCK (simple SELECT .. FROM aTable will be checked against TSQLAccessRights.GET[] per-table right
- reService will indicate the right to execute the interface-based JSON-RPC service implementation
- reUrlEncodedSQL will indicate the right to execute a SQL query encoded at the URI level, for a GET (to be used e.g. with XMLHttpRequest, which forced SentData='' by definition), encoded as sql=..... inline parameter
- reUrlEncodedDelete will indicate the right to delete items using a WHERE clause for DELETE verb at /root/TableName?WhereClause
- reOneSessionPerUser will force that only one session may be created for one user, even if connection comes from the same IP: in this case, you may have to set the SessionTimeOut to a small value, in case the session is not closed gracefully
- by default, read/write access to the TSQLAuthUser table is disallowed, for obvious security reasons: but you can define reUserCanChangeOwnPassword so that the current logged user will be able to change its own password
- order of this set does matter, since it will be stored as a byte value e.g. by TSQLAccessRights.ToString: ensure that new items will always be appended to the list, not inserted within

TSQLAuthGroupClass = class of TSQLAuthGroup;
Class-reference type (metaclass) of the table containing the available user access rights for authentication, defined as a group

TSQLAuthUserClass = class of TSQLAuthUser;
Class-reference type (metaclass) of a table containing the Users registered for authentication
- see also TSQLRestServer.OnAuthenticationUserRetrieve custom event

TSQLCheckTableName = ( ctnNoCheck, ctnMustExist, ctnTrimExisting );
The possible options for handling table names
TSQLEvent = ( seAdd, seUpdate, seDelete, seUpdateBlob );

*Used to define the triggered Event types for TNotifySQLEvent*
- some Events can be triggered via TSQLRestServer.OnUpdateEvent when a Table is modified, and actions can be authorized via overriding the TSQLRest.RecordCanBeUpdated method
- OnUpdateEvent is called BEFORE deletion, and AFTER insertion or update; it should be used only server-side, not to synchronize some clients: the framework is designed around a stateless RESTful architecture (like HTTP/1.1), in which clients ask the server for refresh (see TSQLRestClientURI.UpdateFromServer)
- is used also by TSQLRecord.ComputeFieldsBeforeWrite virtual method

TSQLFieldTables = set of 0..MAX_SQLTABLES-1;

*Used to store bit set for all available Tables in a Database Model*

TSQLFieldType = ( sftUnknown, sftAnsiText, sftUTF8Text, sftEnumerate, sftSet, sftCurrency, sftID, sftRecord, sftBoolean, sftFloat, sftDateTime, sftTimeLog, sftCurrency, sftObject, sftVariant, sftNullable, sftBlob, sftBlobDynArray, sftBlobCustom, sftUTF8Custom, sftMany, sftModTime, sftCreateTime, sftTID, sftRecordVersion, sftSessionUserID, sftDateTimeMS, sftUnixTime, sftUnixMSTime );

*The available types for any SQL field property, as managed with the database driver*
- sftUnknown: unknown or not defined field type
- sftAnsiText: a WinAnsi encoded TEXT, forcing a NOCASE collation (TSQLRecord Delphi property was declared as AnsiString or string before Delphi 2009)
- sftUTF8Text is UTF-8 encoded TEXT, forcing a SYSTEMNOCASE collation, i.e. using UTF8IComp() (TSQLRecord property was declared as RawUTF8, RawUnicode or WideString - or string in Delphi 2009+) - you may inherit from TSQLRecordNoCase to use the NOCASE standard SQLite3 collation
- sftEnumerate is an INTEGER value corresponding to an index in any enumerate Delphi type; storage is an INTEGER value (fast, easy and size efficient); at display, this integer index will be converted into the left-trimmed lowercased chars of the enumerated type text conversion: TOpenType(1) = otDone -> 'Done'
- sftSet is an INTEGER value corresponding to a bitmapped set of enumeration; storage is an INTEGER value (fast, easy and size efficient); displayed as an integer by default, sets with an enumeration type with up to 64 elements is allowed yet (stored as an Int64)
- sftInteger is an INTEGER (Int64 precision, as expected by SQLite3) field
- sftID is an INTEGER field pointing to the ID/RowID of another record of a table, defined by the class type of the TSQLRecord inherited property; coherency is always ensured: after a delete, all values pointing to it is reset to 0
- sftRecord is an INTEGER field pointing to the ID/RowID of another record: TRecordReference=Int64 Delphi property which can be typecasted to RecordRef; coherency is always ensured: after a delete, all values pointing to it are reset to 0 by the ORM
- sftBoolean is an INTEGER field for a boolean value: 0 is FALSE, anything else TRUE (encoded as JSON 'true' or 'false' constants)
- sftFloat is a FLOAT (floating point double precision, cf. SQLite3) field, defined as double (or single) published properties definition
- sftDateTime is a ISO 8601 encoded (SQLite3 compatible) TEXT field, corresponding to a TDateTime Delphi property: a ISO8601 collation is forced for such column, for proper date/time sorting and searching
- sftDateTimeMS is a ISO 8601 encoded (SQLite3 compatible) TEXT field, corresponding to a TDateTimeMS Delphi property, i.e. a TDateTime with millisecond resolution, serialized with '.sss' suffix: a ISO8601 collation is forced for such column, for proper date/time sorting and searching
- sftTimeLog is an INTEGER field for coding a date and time (not SQLite3 compatible), which should be defined as TTimeLog=Int64 Delphi property, ready to be typecasted to the TTimeLogBits
optimized type for efficient timestamp storage, with a second resolution
- `sftCurrency` is a `FLOAT` containing a 4 decimals floating point value, compatible with the Currency Delphi type, which minimizes rounding errors in monetary calculations which may occur with `sftFloat` type
- `sftObject` is a `TEXT` containing an `ObjectToJSON` serialization, able to handle published properties of any not `TPersistent` as `JSON` object, `TStrings` or `TRawUTF8List` as `JSON` arrays of `strings`, `TCollection` or `TObjectList` as `JSON` arrays of `JSON` objects
- `sftVariant` is a `TEXT` containing a variant value encoded as `JSON`: string values are stored between quotes, numerical values directly stored, and `JSON` objects or arrays will be handled as `TDocVariant` custom types
- `sftNullable` is a `INTEGER/DOUBLE/TEXT` field containing a `NULLable` value, stored as a local variant property, identifying `TNullableInteger`, `TNullableBoolean`, `TNullableFloat`, `TNullableCurrency`, `TNullableDateTime`, `TNullableTimeLog` and `TNullableUTF8Text` types
- `sftBlob` is a `BLOB` field (`TSQLRawBlob` Delphi property), and won't be retrieved by default (not part of ORM "simple types"), to save bandwidth
- `sftBlobDynArray` is a dynamic array, stored as `BLOB` field: this kind of property will be retrieved by default, i.e. is recognized as a "simple field", and will use Base64 encoding during `JSON` transmission, or a true `JSON` array, depending on the database back-end (e.g. `MongoDB`)
- `sftBlobCustom` is a custom property, stored as `BLOB` field: such properties are defined by adding a `TSQLPropInfoCustom` instance, overriding `TSQLRecord.InternalRegisterCustomProperties` virtual method - they will be retrieved by default, i.e. recognized as "simple fields"
- `sftUTF8Custom` is a custom property, stored as `JSON` in a `TEXT` field, defined by overriding `TSQLRecord.InternalRegisterCustomProperties` virtual method, and adding a `TSQLPropInfoCustom` instance, e.g. via `RegisterCustomPropertyFromTypeName()` or `RegisterCustomPropertyFromRTTI()`; they will be retrieved by default, i.e. recognized as "simple fields"
- `sftMany` is a 'many to many' field (`TSQLRecordMany` Delphi property); nothing is stored in the table row, but in a separate pivot table: so there is nothing to retrieve here; in contrast to other `TSQLRecord` published properties, which contains an INTEGER ID, the `TSQLRecord.Create` will instantiate a true `TSQLRecordMany` instance to handle this pivot table via its dedicated `ManyAdd/FillMany/ManySelect` methods - as a result, such properties won't be retrieved by default, i.e. not recognized as "simple fields" unless you used the dedicated methods
- `sftModTime` is an `INTEGER` field containing the `TModTime` value, aka time of the record latest update; `TModTime` (just like `TTimeLog` or `TCreateTime`) published property can be typecasted to the `TTimeLogBits` memory structure; the value of this field is automatically updated with the current time and date each time a record is updated (with external DB, it will use the Server time, as retrieved from `SynDB`) - see `ComputeFieldsBeforeWrite` virtual method of `TSQLRecord`; note also that only RESTful PUT/POST access will change this field value: manual SQL statements (like 'UPDATE Table SET Column=Value') won't change its content; note also that this is automated on Delphi client side, so only within `TSQLRecord ORM` use (a pure AJAX application should fill such fields explicitly before sending)
- `sftCreateTime` is an `INTEGER` field containing the `TCreateTime` time of the record creation; `TCreateTime` (just like `TTimeLog` or `TModTime`) published property can be typecasted to the `TTimeLogBits` memory structure; the value of this field is automatically updated with the current date and time when the record is created (with external DB, it will use the Server time, as retrieved from `SynDB`) - see `ComputeFieldsBeforeWrite` virtual method of `TSQLRecord`; note also that only RESTful PUT/POST access will set this field value: manual SQL statements (like 'INSERT INTO Table ...') won't set its content; note also that this is automated on Delphi client side, so only within `TSQLRecord ORM` use (a pure AJAX application should fill such fields explicitly before sending)
- `sftTID` is an `INTEGER` field containing a `TID` pointing to another record; since regular `TSQLRecord` published properties (i.e. `sftID` kind of field) can not be greater than 2,147,483,647 (i.e. a signed
32-bit value) under Win32, defining TID published properties will allow to store the ID as signed 64-bit, e.g. up to 9,223,722,368,854,775,808; despite to sftID kind of record, coherency is NOT ensured: after a deletion, all values pointing to are NOT reset to 0 - it is up to your business logic to ensure data coherency as expected
- sftRecordVersion is an INTEGER field containing a TRecordVersion monotonic number: adding such a published field to any TSQLRecord will allow tracking of record modifications, at storage level; by design, such a field won't be part of "simple types", so won't be transmitted between the clients and the server, but will be updated at any write operation by the low-level Engine(*()) storage methods - such a field will use a TSQLRecordTableDeletion table to track the deleted items
- sftSessionUserID is an INTEGER field containing the TSQLAuthUser.ID of the record modification; the value of this field is automatically updated with the current User ID of the active session; note also that only RESTful PUT/POST access will change this field value: manual SQL statements (like 'UPDATE Table SET Column=0') won't change its content; this is automated on Delphi client side, so only within TSQLRecord ORM use (a pure AJAX application should fill such fields explicitly before sending)
- sftUnixTime is an INTEGER field for coding a date and time as second-based Unix Time (SQLite3 compatible), which should be defined as TUnixTime=Int64 TSQLRecord property
- sftUnixMSTime is an INTEGER field for coding a date and time as millisecond-based Unix Time (JavaScript compatible), which should be defined as TUnixMSTime=Int64 TSQLRecord property
- WARNING: do not change the order of items below, otherwise some methods (like TSQLRecordProperties.CheckBinaryHeader) may be broken and fail

TSQLFieldTypeArray = array[0..MAX_SQLFIELDS] of TSQLFieldType;
/* a fixed array of SQL field property types */

TSQLFieldTypes = set of TSQLFieldType;
Set of available SQL field property types

TSQLHistoryEvent = ( heAdd, heUpdate, heDelete, heArchiveBlob );
Used to define the triggered Event types for TSQLRecordHistory
- TSQLRecordHistory.History will be used for heArchiveBlob
- TSQLRecordHistory.SentDataJSON will be used for other kind of events

TSQLHttpServerRestAuthentication = ( adDefault, adHttpBasic, adWeak, adSSPI );
Supported REST authentication schemes
- used by the overloaded TSQLHttpServer.Create(TSQLHttpServerDefinition) constructor in mORMotHttpServer.pas, and also in dddInfraSettings.pas
- adSSPI won't be defined under Linux, since it is a Windows-centric feature

TSQLInitializeTableOption = ( itoNoAutoCreateGroups, itoNoAutoCreateUsers, itoNoCreateMissingField, itoNoIndex4ID, itoNoIndex4UniqueField, itoNoIndex4NestedRecord, itoNoIndex4RecordReference, itoNoIndex4TID, itoNoIndex4RecordVersion );
The possible options for TSQLRestServer.CreateMissingTables and TSQLRecord.InitializeTable methods
- itoNoAutoCreateGroups and itoNoAutoCreateUsers will avoid TSQLAuthGroup.InitializeTable to fill the TSQLAuthGroup and TSQLAuthUser tables with default records
- itoNoCreateMissingField will avoid to create the missing fields on a table
- itoNoIndex4ID won't create the index for the main ID field (do nothing on SQLite3, by design - but may be used for tables on external databases)
- itoNoIndex4UniqueField won't create indexes for "stored AS_UNIQUE" fields
- itoNoIndex4NestedRecord won't create indexes for TSQLRecord fields
- itoNoIndex4RecordReference won't create indexes for TRecordReference fields
- itoNoIndex4TID won't create indexes for TID fields
- itoNoIndex4RecordVersion won't create indexes for TRecordVersion fields
- INITIALIZE_TABLE_NOINDEX constant contain all itoNoIndex* items

TSQLInitializeTableOptions = set of TSQLInitializeTableOption;

The options to be specified for TSQLRestServer.CreateMissingTables and TSQLRecord.InitializeTable methods

TSQLListLayout = ( llLeft, llUp, llClient, llLeftUp );

Defines the way the TDrawGrid is displayed by User Interface generation

TSQLLog = TSynLog;

Logging class with enhanced RTTI
- will write TObject/TSQLRecord, enumerations and sets content as JSON
- is the default logging family used by the mORMot framework
- mORMotDB.pas unit will set SynDBLog := TSQSLLog
- mORMotSQLite3.pas unit will set SynSQLite3Log := TSQSLLog

TSQLModelRecordPropertiesObjArray = array of TSQLModelRecordProperties;

Dynamic array of TSQLModelRecordProperties
- used by TSQLModel to store the non-shared information of all its tables

TSQLMonitorUsageClass = class of TSQLMonitorUsage;

Class-reference type (metaclass) of a TSQLMonitorUsage table

TSQLOccasion = ( soSelect, soInsert, soUpdate, soDelete );

Used to defined the CRUD associated SQL statement of a command
- used e.g. by TSQLRecord.GetJSONValues methods and SimpleFieldsBits[] array (in this case, soDelete is never used, since deletion is global for all fields)
- also used for cache content notification

TSQLOccasions = set of TSQLOccasion;

Used to defined a set of CRUD associated SQL statement of a command

TSQLPropInfoAttribute = ( aIsUnique, aAuxiliaryRTreeField, aBinaryCollation );

ORM attributes for a TSQLPropInfo definition

TSQLPropInfoAttributes = set of TSQLPropInfoAttribute;

Set of ORM attributes for a TSQLPropInfo definition

TSQLPropInfoClass = class of TSQLPropInfo;

Class-reference type (metaclass) of a TSQLPropInfo information

TSQLPropInfoListOptions = set of ( pilRaiseEORMExceptionIfNotHandled, pilAllowIDFields, pilSubClassesFlattening, pilIgnoreIfGetter, pilSingleHierarchyLevel, pilAuxiliaryFields);

Define how the published properties RTTI is to be interpreted
- i.e. how TSQLPropInfoList.Create() and TSQLPropInfoRTTI.CreateFrom() will handle the incoming RTTI

TSQLPropInfoObjArray = array of TSQLPropInfo;

Dynamic array of ORM fields information for published properties

TSQLPropInfoRTTIClass = class of TSQLPropInfoRTTI;
Class-reference type (metaclass) of a TSQLPropInfoRTTI information

TSQLPropInfoRTTIPtrInt = TSQLPropInfoRTTIInt32;

Information about a IntPtr published property, according to the native CPU
- not a real stand-alone class, but a convenient wrapper type
TSQLQueryEvent = function(aTable: TSQLRecordClass; aID: TID; FieldType: TSQLFieldType; Value: PUTF8Char; Operator: integer; Reference: PUTF8Char): boolean of object;

User Interface Query action evaluation function prototype
- Operator is ord(TSQLQueryOperator) by default (i.e. for class function TSQLRest.QueryIsTrue), or is a custom enumeration index for custom queries (see TSQLQueryCustom.EnumIndex below, and TSQLRest.QueryAddCustom() method)
- for default Operator as ord(TSQLQueryOperator), qoContains and qoBeginWith expect the Reference to be already uppercase
- qoEqualTo to qoGreaterThanOrEqualTo apply to all field kind (work with either numeric either UTF-8 values)
- qoEqualToWithCase to qoSoundsLikeSpanish handle the field as UTF-8 text, and make the comparison using the phonetic algorithm corresponding to a language family
- for default Operator as ord(TSQLQueryOperator), qoSoundsLike* operators expect the Reference not to be a PUTF8Char, but a typcast of a prepared TSynSoundEx object instance (i.e. pointer(@SoundEx)) by the caller
- for custom query (from TSQLQueryCustom below), the event must handle a special first call with Value=nil to select if this custom Operator/Query is available for the specified aTable: in this case, returning true indicates that this custom query is available for this table
- for custom query (from TSQLQueryCustom below), the event is called with FieldType := TSQLFieldType(TSQLQueryCustom.EnumIndex)+64
TSQLQueryOperator = ( qoNone, qoEqualTo, qoNotEqualTo, qoLessThan, qoLessThanOrEqualTo, qoGreaterThan, qoGreaterThanOrEqualTo, qoContains, qoBeginWith, qoSoundsLikeEnglish, qoSoundsLikeFrench, qoSoundsLikeSpanish );

UI Query comparison operators
- these operators are e.g. used to mark or unmark some lines in a UI Grid or for TInterfaceStub.ExpectsCount() methods
TSQLQueryOperators = set of TSQLQueryOperator;

Set of UI Query comparison operators
TSQLRawBlob = type RawByteString;

A String used to store the BLOB content
- equals RawByteString for byte storage, to force no implicit charset conversion, whatever the codepage of the resulting string is
- will identify a sftBlob field type, if used to define such a published property
- by default, the BLOB fields are not retrieved or updated with raw TSQLRest.Retrieve() method, that is "Lazy loading" is enabled by default for blobs, unless TSQLRestClientURI.ForceBlobTransfert property is TRUE (for all tables), or ForceBlobTransfertTable[] (for a particular table); so use RetrieveBlob() methods for handling BLOB fields
- could be defined as value in a TSQLRecord property as such:

property Blob: TSQLRawBlob read fBlob write fBlob;

TSQLRecordClass = class of TSQLRecord;

Class-reference type (metaclass) of TSQLRecord
TSQLRecordClassDynArray = array of TSQLRecordClass;  
A dynamic array used to store the TSQLRecord classes in a Database Model

TSQLRecordFTS3Class = class of TSQLRecordFTS3;  
Class-reference type (metaclass) of a FTS3/FTS4/FTS5 virtual table

TSQLRecordHistoryClass = class of TSQLRecordHistory;  
Class-reference type (metaclass) to specify the storage table to be used for tracking TSQLRecord changes  
- you can create your custom type from TSQLRecordHistory, even for a particular table, to split the tracked changes storage in several tables:  
  type  
  TSQLRecordMyHistory = class(TSQLRecordHistory);  
- as expected by TSQLRestServer.TrackChanges() method

TSQLRecordManyJoinKind = ( jkDestID, jkPivotID, jkDestFields, jkPivotFields, jkPivotAndDestFields );  
The kind of fields to be available in a Table resulting of a TSQLRecordMany.DestGetJoinedTable() method call  
- Source fields are not available, because they will be always the same for a same SourceID, and they should be available from the TSQLRecord which hold the TSQLRecordMany instance  
- jkDestID and jkPivotID will retrieve only DestTable.ID and PivotTable.ID  
- jkDestFields will retrieve DestTable.* simple fields, or the fields specified by aCustomFieldsCSV (the Dest table name will be added: e.g. for aCustomFieldsCSV='One,Two', will retrieve DestTable.One, DestTable.Two)  
- jkPivotFields will retrieve PivotTable.* simple fields, or the fields specified by aCustomFieldsCSV (the Pivot table name will be added: e.g. for aCustomFieldsCSV='One,Two', will retrieve PivotTable.One, PivotTable.Two)  
- jkPivotAndDestAllFields for PivotTable.* and DestTable.* simple fields, or will retrieve the specified aCustomFieldsCSV fields (with the table name associated: e.g. 'PivotTable.One, DestTable.Two')

TSQLRecordManyObjArray = array of TSQLRecordMany;  
A dynamic array of TSQLRecordMany instances

TSQLRecordObjArray = array of TSQLRecord;  
A dynamic array storing TSQLRecord instances  
- not used directly, but as specialized T*ObjArray types

TSQLRecordPropertiesMappingOptions = set of ( rpmAutoMapKeywordFields, rpmNoCreateMissingTable, rpmNoCreateMissingField, rpmMissingFieldNameCaseSensitive, rpmQuoteFieldName, rpmClearPoolOnConnectionIssue);  
Used by TSQLRecordPropertiesMapping.Options for custom field mapping of a TSQLRecord on an external database process  
- rpmAutoMapKeywordFields is set if MapAutoKeywordFields has been defined, i.e. if field names which may conflict with a keyword should be automatically mapped to a harmless symbol name  
- rpmNoCreateMissingTable will bypass the existing table check, e.g. to circumvent some specific DB provider or case sensitivity issue on tables  
- rpmNoCreateMissingField will bypass the existing field check, e.g. to circumvent some specific DB provider or case sensitivity issue on fields  
- by default, check of missing field name will be case insensitive, unless the rpmMissingFieldNameCaseSensitive option is set  
- rpmQuoteFieldName will quote the field names - to be used e.g. with FireBird in its Dialect 3  
- rpmClearPoolOnConnectionIssue will enable detecting connection loss
**TSQLRecordRTreeClass** = class of TSQLRecordRTreeAbstract;  
*Class-reference type (metaclass) of a RTREE virtual table*  
either a TSQLRecordRTree or a TSQLRecordRTreeInteger

**TSQLRecordServiceLogClass** = class of TSQLRecordServiceLog;  
*Class-reference type (metaclass) for storing interface-based service execution statistics*  
you could inherit from TSQLRecordServiceLog, and specify additional fields corresponding to the execution context

**TSQLRecordServiceNotificationsClass** = class of TSQLRecordServiceNotifications;  
*Class-reference type (metaclass) for storing interface-based service execution statistics used for DB-based asynchronous notifications*  
as used by TServiceFactoryClient.SendNotifications

**TSQLRecordTableDeletedClass** = class of TSQLRecordTableDeleted;  
*Class-reference type (metaclass) to specify the storage table to be used for tracking TSQLRecord deletion*

**TSQLRecordTreeCoords** = array[0..RTREE_MAX_DIMENSION-1] of packed record min, max: double;  
*This kind of record array can be used for direct floating-point coordinates storage as in TSQLRecordRTree.BlobToCoord*

**TSQLRecordTreeCoordsInteger** = array[0..RTREE_MAX_DIMENSION-1] of packed record min, max: integer;  
*This kind of record array can be used for direct 32-bit integer coordinates storage as in TSQLRecordRTreeInteger.BlobToCoord*

**TSQLRecordVirtualKind** = ( rSQLite3, rFTS3, rFTS4, rFTS5, rRTree, rRTreeInteger,  
rCustomForcedID, rCustomAutoID );  
The kind of SQLite3 (virtual) table  
TSQLRecordFTS3/4/5 will be associated with vFTS3/vFTS4/vFTS5 values,  
TSQLRecordRTree/TSQLRecordRTreeInteger with rRTree/rRTreeInteger, any native SQLite3 table as vSQLite3, and a TSQLRecordVirtualTable*ID as rCustomForcedID/rCustomAutoID  
a plain TSQLRecord class can be defined as rCustomForcedID (e.g. for TSQLRecordMany) after registration for an external DB via a call to VirtualTableExternalRegister() from mORMotDB unit

**TSQLRestBatchOption** = ( boInsertOrIgnore, boInsertOrUpdate, boExtendedJSON,  
boPostNoSimpleFields, boPutNoCacheFlush, boRollbackOnError );  
The available options for TSQLRest.BatchStart() process  
- boInsertOrIgnore will create 'INSERT OR IGNORE' statements instead of plain 'INSERT' - by now, only the direct mORMotSQLite3 engine supports it  
- boInsertOrUpdate will create 'INSERT OR REPLACE' statements instead of plain 'INSERT' - by now, only the direct mORMotSQLite3 engine supports it  
- boExtendedJSON will force the JSON to unquote the column names, e.g. writing col1:...,col2:...  
  instead of "col1":...,"col2"...  
- boPostNoSimpleFields will avoid to send a TSQLRestBatch.Add() with simple fields as  
  "SIMPLE":[val1,val2...] or "SIMPLE@tablename":[val1,val2...], without the field names  
- boPutNoCacheFlush won't force the associated Cache entry to be flushed: it is up to the caller to ensure cache coherency  
- boRollbackOnError will raise an exception and Rollback any transaction if any step failed - default if to continue batch processs, but setting a value <> 200/HTTP_SUCCESS in Results[]

**TSQLRestBatchOptions** = set of TSQLRestBatchOption;
A set of options for TSQLRest.BatchStart() process

- TJSONObjectDecoder will use it to compute the corresponding SQL

TSQLRestCacheEntryDynArray = array of TSQLRestCacheEntry;

For TSQLRestCache, stores all table settings and values

- this dynamic array will follow TSQLRest.Model.Tables[] layout, i.e. one entry per TSQLRecord class in the data model

TSQLRestCacheEntryValueDynArray = array of TSQLRestCacheEntryValue;

For TSQLRestCache, stores all tables values

TSQLRestClass = class of TSQLRest;

Class-reference type (metaclass) of a TSQLRest kind

TSQLRestDynArray = array of TSQLRest;

A dynamic array of TSQLRest instances

TSQLRestModelMatch = (rmNoMatch, rmMatchExact, rmMatchWithCaseChange);

How TSQLModel.URIMatch() will compare an URI

- will allow to make a difference about case-sensitivity

TSQLRestObjArray = array of TSQLRest;

A dynamic array of TSQLRest instances, owning the instances

TSQLRestServerAcquireMode = (amUnlocked, amLocked, amBackgroundThread, amBackgroundORMSharedThread, amMainThread);

How a TSQLRest class may execute read or write operations

- used e.g. for TSQLRestServer.AcquireWriteMode or TSQLRestServer.AcquireExecutionMode/AcquireExecutionLockedTimeOut

TSQLRestServerAddStat = (withTables, withMethods, withInterfaces, withSessions);

The flags used for TSQLRestServer.AddStats

TSQLRestServerAddStats = set of TSQLRestServerAddStat;

Some flags used for TSQLRestServer.AddStats

TSQLRestServerAuthenticationClass = class of TSQLRestServerAuthentication;

Class-reference type (metaclass) used to define an authentication scheme

TSQLRestServerAuthenticationClientSetUserPassword = (passClear, passHashed, passKerberosSPN);

Define how TSQLRestServerAuthentication.ClientSetUser() should interpret the supplied password

- passClear means that the password is not encrypted, e.g. as entered by the user in the login screen
- passHashed means that the password is already hashed as in TSQLAuthUser.PasswordHashHexa i.e. SHA256('salt'+Value)
- passKerberosSPN indicates that the password is the Kerberos SPN domain

TSQLRestServerAuthenticationDynArray = array of TSQLRestServerAuthentication;

Maintain a list of TSQLRestServerAuthentication instances

TSQLRestServerAuthenticationOption = (saoUserByLogonOrID, saoHandleUnknownLogonAsStar);

Optional behavior of TSQLRestServerAuthentication class

- by default, saoUserByLogonOrID is set, allowing TSQLRestServerAuthentication.GetUser() to retrieve the TSQLAuthUser by logon name or by ID, if the supplied logon name is an integer
if saoHandleUnknownLogonAsStar is defined, any user successfully authenticated could be logged with the same ID (and authorization) than TSQLAuthUser.Logon="*" - of course, this is meaningfull only with an external credential check (e.g. via SSPI or Active Directory)

```
TSQLRestServerAuthenticationOptions = set of TSQLRestServerAuthenticationOption;

Defines the optional behavior of TSQLRestServerAuthentication class
```

```
TSQLRestServerAuthenticationSignedURIAlgo = ( suaCRC32, suaCRC32C, suaXXHASH, suaMD5, suaSHA1, suaSHA256, suaSHA512 );

Algorithms known by TSQLRestServerAuthenticationSignedURI to digitally compute the session_signature parameter value for a given URI
- by default, suaCRC32 will compute fast but not cryptographically secure
  crc32(crc32(private_salt,timestamp,8),url,urllen)
- suaCRC32C and suaXXHASH will be faster and slightly safer
- but you can select other stronger alternatives, which result will be reduced to 32-bit hexadecimal - suaMD5 will be the fastest cryptographic hash available on all platforms, for enhanced security, by calling e.g.
  (aServer.AuthenticationRegister(TSQLRestServerAuthenticationDefault) as TSQLRestServerAuthenticationDefault).Algorithm := suaMD5;
- suaSHA1, suaSHA256 and suaSHA512 will be the slowest, to provide additional level of trust, depending on your requirements: note that since the hash is reduced to 32-bit resolution, those may not provide higher security than suaMD5
- note that SynCrossPlatformRest clients only implements suaCRC32 yet

TSQLRestServerAuthenticationSignedURIComputeSignature = function( privatesalt: cardinal; timestamp, url: PAnsiChar; urllen: integer): cardinal of object;

Function prototype for TSQLRestServerAuthenticationSignedURI computation of the session_signature parameter value
```

```
TSQLRestServerCallBack = procedure(Ctxt: TSQLRestServerURIContext) of object;

Method prototype to be used on Server-Side for method-based services
- will be routed as ModelRoot/[TableName/TableID/]MethodName RESTful requests
- this mechanism is able to handle some custom Client/Server request, similar to the DataSnap technology, but in a KISS way: it's fully integrated in the Client/Server architecture of our framework
- just add a published method of this type to any TSQLRestServer descendant
- when TSQLRestServer.URI receive a request for ModelRoot/MethodName or ModelRoot/TableName/TableID/MethodName, it will check for a published method in its self instance named MethodName which MUST be of TSQLRestServerCallBack type (not checked neither at compile time neither at runtime: beware!) and call it to handle the request
- important warning: the method implementation MUST be thread-safe
- when TSQLRestServer.URI receive a request for ModelRoot/MethodName, it calls the corresponding published method with aRecord set to nil
- when TSQLRestServer.URI receive a request for ModelRoot/TableName/TableID/MethodName, it calls the corresponding published method with aRecord pointing to a just created instance of the corresponding class, with its field ID set; note that the only set field is ID: other fields of aRecord are not set, but must seculfaly be retrieved on purpose
- for ModelRoot/TableName/TableID/MethodName, the just created instance will be freed by TSQLRestServer.URI when the method returns
- Ctxt.Parameters values are set from incoming URI, and each parameter can be retrieved with a loop like this:

  ```
  if not UrlDecodeNeedParameters(Ctxt.Parameters,'SORT,COUNT') then exit;
  while Ctxt.Parameters<>nil do begin
  ```
UrlDecodeValue(Ctxt.Parameters, 'SORT=', aSortString);
UrlDecodeValueInteger(Ctxt.Parameters, 'COUNT=', aCountInteger, @Ctxt.Parameters);
end;

- Ctx.Call is set with low-level incoming and outgoing data from client (e.g. Ctx.Call.InBody contain POST/PUT data message)
- Ctx.Session* will identify to the authentication session of the remote client
  (CONST_AUTHENTICATION_NOT_USED=1 if authentication mode is not enabled or
  CONST_AUTHENTICATION_SESSION_NOT_STARTED=0 if the session not started yet) - code may use
  SessionGetUser() method to retrieve the user details
- Ctx.Method will indicate the used HTTP verb (e.g. GET/POST/PUT..)
- if process succeeded, implementation shall call Ctx.Results([]) method to set a JSON response
  object with one "result" field name or Ctx.Returns() and its optional
  CustomHeader parameter can specify a custom header like TEXT_CONTENT_TYPE_HEADER
- if process succeeded, and no data is expected to be returned to the caller, implementation shall call
  overloaded Ctx.Success() method with the expected status (i.e. just Ctx.Success will return
  HTTP_SUCCESS)
- if process failed, implementation shall call Ctx.Error() method to set the corresponding error
  message and error code number
- a typical implementation may be:

```pascal
procedure TSQLRestServerTest.Sum(Ctxt: TSQLRestServerURIContext);
var
  a, b: TSynExtended;
begin
  if UrlDecodeNeedParameters(Ctxt.Parameters, 'A,B') then begin
    while Ctxt.Parameters<>nil do begin
      UrlDecodeExtended(Ctxt.Parameters, 'A=', a);
      UrlDecodeExtended(Ctxt.Parameters, 'B=', b, @Ctxt.Parameters);
    end;
    Ctxt-results[a+b];
    // same as: Ctx.Returns(JSONEncode(['result',a+b]));
    // same as: Ctx.Returns(['result',a+b]);
  end else
    Ctx.Error('Missing Parameter');
end;
```

- Client-Side can be implemented as you wish. By convention, it could be appropriate to define in
  either TSQLRestServer (if to be called as ModelRoot/MethodName), either TSQLRecord (if to be
  called as ModelRoot/TableName[/TableID]/MethodName) a custom public or protected method,
  calling TSQLRestClientURI.URL with the appropriate parameters, and named (by convention) as
  MethodName; TSQLRestClientURI has dedicated methods like CallBackGetResult, CallBackGet,
  CallBackPut and CallBack; see also TSQLModel.getURI and JSONDecode function

```pascal
function TSQLRecordPeople.Sum(aClient: TSQLRestClientURI; a, b: double): double;
var
  err: integer;
begin
  try
    val(aClient.CallBackGetResult('sum', ['a', a, 'b', b]), result, err);
  end;
end;
```

TSQLRestServerClass = class of TSQLRestServer;

Class-reference type (metaclass) of a REST server

TSQLRestServerKind = ( sMainEngine, sStaticDataTable, sVirtualTable );

Kind of (static) database server implementation available
- sMainEngine will identify the default main SQLite3 engine
- sStaticDataTable will identify a TSQLRestStorageInMemory - i.e. TSQLRestServer.fStaticData[] which
  can work without SQLite3
- sVirtualTable will identify virtual TSQLRestStorage classes - i.e. TSQLRestServer.fStaticVirtualTable[]
which points to SQLite3 virtual tables (e.g. TObjectList or external databases)

```pascal
TSQLRestServerMethods = array of TSQLRestServerMethod;
```

*Used to store all method-based services of a TSQLRestServer instance*

```pascal
TSQLRestServerMonitorLevels = set of ( mlTables, mlMethods, mlInterfaces, mlSessions, mSQLite3);
```

*How TSQLRestServer should maintain its statistical information*
- used by TSQLRestServer.StatLevels property

```pascal
TSQLRestServerNotifyCallback = function(aSender: TSQLRestServer; const aInterfaceDotMethodName,aParams: RawUTF8; aConnectionID: Int64; aFakeCallID: integer; aResult, aErrorMsg: PRawUTF8): boolean of object;
```

*Event signature used to notify a client callback*
- implemented e.g. by TSQLHttpServer.NotifyCallback

```pascal
TSQLRestServerOption = ( rsoNoAJAXJSON, rsoGetAsJsonNotAsString, rsoGetID_str, rsoRedirectForbiddenToAuth, rsoHttp200WithNoBodyReturns204, rsoAddUpdateReturnsContent, rsoComputeFieldsBeforeWriteOnServerSide, rsoSecureConnectionRequired, rsoCookieIncludeRootPath, rsoCookieHttpOnlyFlagDisable, rsoAuthenticationURIDisable, rsoTimestampInfoURIDisable, rsoHttpHeaderCheckDisable, rsoGetUserRetrieveNoBlobData, rsoNoInternalState );
```

*Some options for TSQLRestServer process*
- read-only rsoNoAJAXJSON indicates that JSON data is transmitted in "not expanded" format: you should NEVER change this option by including this property in TSQLRestServer.Options, but always call explicitly TSQLRestServer.NoAJAXJSON := true so that the SetNoAJAXJSON virtual method should be called as expected (e.g. to flush TSQLRestServerDB cache)
- rsoGetAsJsonNotAsString will let ORM GET return to AJAX (non Delphi) clients JSON objects instead of the JSON text stored in database fields
- rsoGetID_str will add a "ID_str": string field to circumvent JavaScript limitation of 53-bit for integers
- only for AJAX (non Delphi) clients
- unauthenticated requests from browsers (i.e. not Delphi clients) may be redirected to the TSQLRestServer.Auth() method via rsoRedirectForbiddenToAuth (e.g. for TSQLRestServerAuthenticationHttpBasic popup)
- some REST/AJAX clients may expect to return status code 204 as instead of 200 in case of a successful operation, but with no returned body (e.g. a DELETE with SAPUI5 / OpenUI5 framework): include rsoHttp200WithNoBodyReturns204 so that any HTTP_SUCCESS (200) with no returned body will return a HTTP_NOCONTENT (204), as expected by some clients
- by default, Add() or Update() will return HTTP_CREATED (201) or HTTP_SUCCESS (200) with no body, unless rsoAddUpdateReturnsContent is set to return as JSON the last inserted/updated record
- TModTime / TCreateTime fields are expected to be filled on client side, unless you set rsoComputeFieldsBeforeWriteOnServerSide so that AJAX requests will set the fields on the server side by calling the TSQLRecord ComputeFieldsBeforeWrite virtual method, before writing to the database
- rsoSecureConnectionRequired will ensure Call is flagged as llfSecured (i.e. in-process, HTTPS, or encrypted WebSockets) - with the only exception of the Timestamp method-based service (for monitoring purposes) - note that this option doesn't make sense behind a proxy, just with a true HTTPS server
- by default, cookies will contain only 'Path=/Model.Root', but '; Path=/' may be also added setting rsoCookieIncludeRootPath
- you can disable the 'HttpOnly' flag via rsoCookieHttpOnlyFlagDisable
- TSQLRestServerURIContext.AuthenticationBearerToken will return the ?authenticationbearer=... URI parameter value alternatively to the HTTP header unless rsoAuthenticationURIDisable is set (for...
Synopsys mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

security reasons)
- you can switch off root/timestamp/info URI via rsoTimestampInfoURIDisable
- URI() header output will be sanitized for any EOL injection, unless rsoHttpHeaderCheckDisable is defined (to gain a few cycles?)
- by default, TSQLAuthUser.Data blob is retrieved from the database, unless rsoGetUserRetrieveNoBlobData is defined
- rsoNoInternalState could be state to avoid transmitting the 'Server-InternalState' header, e.g. if the clients wouldn't need it

TSQLRestServerOptions = set of TSQLRestServerOption;
Allow to customize the TSQLRestServer process via its Options property

TSQLRestServerURIContextClass = class of TSQLRestServerURIContext;
Class used to define the Client-Server expected routing
- most of the internal methods are declared as virtual, so it allows any kind of custom routing or execution scheme
- TSQLRestRoutingREST and TSQLRestRoutingJSON_RPC classes are provided in this unit, to allow RESTful and JSON/RPC protocols

TSQLRestServerURIContextClientInvoke = set of (csiAsOctetStream);
/used to customize TSQLRestServerURIContext.ClientSideInvoke process

TSQLRestServerURIContextClientKind = ( ckUnknown, ckFramework, ckAJAX );
Used by TSQLRestServerURIContext.ClientKind to identify the currently connected client

TSQLRestServerURIContextCommand = ( execNone, execSOAByMethod, execSOAByInterface, execORMGet, execORMWrite );
All commands which may be executed by TSQLRestServer.URI() method
- execSOAByMethod for method-based services
- execSOAByInterface for interface-based services
- execORMGet for ORM reads i.e. Retrieve*() methods
- execORMWrite for ORM writes i.e. Add Update Delete TransactionBegin Commit Rollback methods

TSQLRestServerURIDynArray = array of TSQLRestServerURI;
Store a list of TSQLRestServer URIs

TSQLRestServerURIString = type RawUTF8;
A specialized UTF-8 string type, used for TSQLRestServerURI storage
- URI format is 'address:port/root', but port or root are optional
- you could use TSQLRestServerURI record to store and process it

TSQLRestServerURIStringDynArray = array of TSQLRestServerURIString;
A list of UTF-8 strings, used for TSQLRestServerURI storage
- URI format is 'address:port/root', but port or root are optional
- you could use TSQLRestServerURI record to store and process each item

TSQLRestStorageClass = class of TSQLRestStorage;
Class-reference type (metaclass) of our abstract table storage
- may be e.g. TSQLRestStorageInMemory, TSQLRestStorageInMemoryExternal, TSQLRestStorageExternal or TSQLRestStorageMongoDB

TSQLRestStorageInMemoryClass = class of TSQLRestStorageInMemory;
Class-reference type (metaclass) of our TObjectList memory-stored table storage
- may be TSQLRestStorageInMemory or TSQLRestStorageInMemoryExternal
TSQLRestStorageInMemoryDynArray = array of TSQLRestStorageInMemory;
A dynamic array of TSQLRestStorageInMemory instances
- used e.g. by TSQLRestServerFullMemory

TSQLRestStorageShardClass = class of TSQLRestStorageShard;
Class metadata of a Sharding storage engine

TSQLRestStorageShardOption = ( ssoNoUpdate, ssoNoUpdateButLastShard, ssoNoDelete,
ssoNoDeleteButLastShard, ssoNoBatch, ssoNoList, ssoNoExecute, ssoNoUpdateField,
ssoNoConsolidateAtDestroy );
Defines how TSQLRestStorageShard will handle its partitioned process

TSQLRestStorageShardOptions = set of TSQLRestStorageShardOption;
How TSQLRestStorageShard will handle its partitioned process

TSQLRestTempStorageItemDynArray = array of TSQLRestTempStorageItem;
Used to store the entries in the TSQLRestTempStorage class

TSQLRestTempStorageItemKind = set of (itemInsert, itemFakeID);
Defines what is stored in a TSQLRestTempStorageItem entry

TSQLRestURIParamsLowLevelFlag = ( llfHttps, llfSecured, llfWebsockets );
Flags which may be set by the caller to notify low-level context
- llfHttps will indicate that the communication was made over HTTPS
- llfSecured is set if the transmission is encrypted or in-process, using e.g. HTTPS/SSL/TLS or our proprietary AES/ECDHE algorithms
- llfWebsockets communication was made using WebSockets

TSQLRestURIParamsLowLevelFlags = set of TSQLRestURIParamsLowLevelFlag;
Some flags set by the caller to notify low-level context

TSQLURIMethod = ( mNone, mGET, mPOST, mPUT, mDELETE, mHEAD, mBEGIN, mEND, mABORT, mLOCK,
mUNLOCK, mSTATE, mOPTIONS, mPROPFIND, mPROPPATCH, mTRACE, mCOPY, mMKCOL, mMOVE, mPURGE,
mREPORT, mMKACTIVITY, mMKCALENDAR, mCHECKOUT, mMERGE, mNOTIFY, mPATCH, mSEARCH,
mCONNECT );
The available HTTP methods transmitted between client and server
- some custom verbs are available in addition to standard REST commands
- most of iana verbs are available see http://www.iana.org/assignments/http-methods/http-methods.xhtml
- for basic CRUD operations, we consider Create=mPOST, Read=mGET, Update=mPUT and Delete=mDELETE - even if it is not fully RESTful

TSQLURIMethods = set of TSQLURIMethod;
Set of available HTTP methods transmitted between client and server

TSQLVirtualTableClass = class of TSQLVirtualTable;
Class-reference type (metaclass) of a virtual table implementation

TSQLVirtualTableCursorClass = class of TSQLVirtualTableCursor;
Class-reference type (metaclass) of a cursor on an abstract Virtual Table

TSQLVirtualTableFeature = ( vtWrite, vtTransaction, vtSavePoint, vtWhereIDPrepared );
The possible features of a Virtual Table
- vtWrite is to be set if the table is not Read/Only
- vtTransaction if handles vttBegin, vttSync, vttCommit, vttRollback
- vtSavePoint if handles vttSavePoint, vttRelease, vttRollBackTo
- vtWhereIDPrepared if the ID=? WHERE statement will be handled in TSQLVirtualTableCursor.Search()

TSQLVirtualTableFeatures = set of TSQLVirtualTableFeature;
A set of features of a Virtual Table

TSQLVirtualTablePreparedCost = ( costFullScan, costScanWhere, costSecondaryIndex, costPrimaryIndex );
Abstract planning execution of a query, as set by TSQLVirtualTable.Prepare

TSQLVirtualTableTransaction = ( vttBegin, vttSync, vttCommit, vttRollBack, vttSavePoint, vttRelease, vttRollBackTo );
The available transaction levels

TSynMonitorUsageGranularities = set of TSynMonitorUsageGranularity;
Defines one or several time periods for TSynMonitorUsage process

TSynMonitorUsageGranularity = ( mugUndefined, mugMinute, mugHour, mugDay, mugMonth, mugYear );
The time periods covered by TSynMonitorUsage process
- defines the resolution of information computed and stored

TURIMapRequest = function(url, method, SendData: PUTF8Char; Resp, Head: PPUTF8Char): Int64Rec; cdecl;
Function prototype for remotely calling a TSQLRestServer
- use PUTF8Char instead of string: no need to share a memory manager, and can be used with any language (even C or .NET, thanks to the cdecl calling convention)
- you can specify some POST/PUT data in SendData (leave as nil otherwise)
- returns in result.Lo the HTTP STATUS integer error or success code
- returns in result.Hi the server database internal status
- on success, allocate and store the resulting JSON body into Resp^, headers in Head^
- use a GlobalFree() function to release memory for Resp and Head responses

Constants implemented in the mORMot unit

ALL_ACCESS_RIGHTS = [0..MAX_SQLTABLES-1];
Supervisor Table access right, i.e. allmighty over all fields

ALL_FIELDS: TSQLFieldBits = [0..MAX_SQLFIELDS-1];
Special TSQLFieldBits value containing all field bits set to 1

AS_UNIQUE = false;
Used as "stored AS_UNIQUE" published property definition in TSQLRecord

CONST_AUTHENTICATION_NOT_USED = 1;
The used TAuthSession.IDCardinal value if authentication mode is not set
- i.e. if TSQLRest.HandleAuthentication equals FALSE

CONST_AUTHENTICATION_SESSION_NOT_STARTED = 0;
The used TAuthSession.IDCardinal value if the session not started yet
- i.e. if the session handling is still in its handshaking phase

COOKIE_EXPIRED = '; Expires=Sat, 01 Jan 2010 00:00:01 GMT';
You can use this cookie value to delete a cookie on the browser side
**COPIABLE_FIELDS**: `TSQLFieldTypes = [low(TSQLFieldType)..high(TSQLFieldType)] - [sftUnknown, sftMany];

Kind of fields which can be copied from one TSQLRecord instance to another

**DEFAULT_HASH_SYNOPSE** = '67aeea294e1cb515236fd7829c55ec820ef888e8e221814d24d83b3dc4d825dd';

Default hashed password set by TSQLAuthGroup.InitializeTable for all users
- contains TSQLAuthUser.ComputeHashedPassword('synopse')
- override AuthAdminDefaultPassword, AuthSupervisorDefaultPassword and AuthUserDefaultPassword values to follow your own application expectations

**FULL_ACCESS_RIGHTS**: `TSQLAccessRights = (AllowRemoteExecute: [reSQL, reSQLSelectWithoutTable, reService, reUrlEncodedSQL, reUrlEncodedDelete]; GET: ALL_ACCESS_RIGHTS; POST: ALL_ACCESS_RIGHTS; PUT: ALL_ACCESS_RIGHTS; DELETE: ALL_ACCESS_RIGHTS);`

Complete Database access right, i.e. allmighty over all Tables
- WITH the possibility to remotely execute any SQL statement (reSQL right)
- is used by default by TSQLRestClientDB.URI() method, i.e. for direct local/in-process access
- is used as reference to create TSQLAuthUser 'Admin' access policy

**GUID_FAKETYPEINFO**: `packed record Kind: TTypeKind; Name: string[5]; Size: cardinal; Count: integer; end = ( Kind: tkRecord; Name: 'TGUID'; Size: SizeOf(TGUID); Count: 0);`

Fake TTypeInfo RTTI used for TGUID on older versions of the compiler

**HTTP_ACCEPTED** = 202;

HTTP Status Code for "Accepted"

**HTTP_BADGATEWAY** = 502;

HTTP Status Code for "Bad Gateway"

**HTTP_BADREQUEST** = 400;

HTTP Status Code for "Bad Request"

**HTTP_CONFLICT** = 409;

HTTP Status Code for "Conflict"

**HTTP_CONTINUE** = 100;

HTTP Status Code for "Continue"

**HTTP_CREATED** = 201;

HTTP Status Code for "Created"

**HTTP_FORBIDDEN** = 403;

HTTP Status Code for "Forbidden"

**HTTP_FOUND** = 302;

HTTP Status Code for "Found"

**HTTP_GATEWAYTIMEOUT** = 504;

HTTP Status Code for "Gateway Timeout"

**HTTP_HTTPVERSIONNONSUPPORTED** = 505;

HTTP Status Code for "HTTP Version Not Supported"

**HTTP_MOVEDPERMANENTLY** = 301;

HTTP Status Code for "Moved Permanently"
HTTP_MULTIPLECHOICES = 300;
   HTTP Status Code for "Multiple Choices"

HTTP_NOCONTENT = 204;
   HTTP Status Code for "No Content"

HTTP_NONAUTHORIZEDINFO = 203;
   HTTP Status Code for "Non-Authoritative Information"

HTTP_NONE = 0;
   Void HTTP Status Code (not a standard value, for internal use only)

HTTP_NOTACCEPTABLE = 406;
   HTTP Status Code for "Not Acceptable"

HTTP_NOTALLOWED = 405;
   HTTP Status Code for "Method Not Allowed"

HTTP_NOTFOUND = 404;
   HTTP Status Code for "Not Found"

HTTP_NOTIMPLEMENTED = 501;
   HTTP Status Code for "Not Implemented"

HTTP_NOTMODIFIED = 304;
   HTTP Status Code for "Not Modified"

HTTP_PARTIALCONTENT = 206;
   HTTP Status Code for "Partial Content"

HTTP_PAYLOADTOOLARGE = 413;
   HTTP Status Code for "Payload Too Large"

HTTP_PROXYAUTHREQUIRED = 407;
   HTTP Status Code for "Proxy Authentication Required"

HTTP_RESETCONTENT = 205;
   HTTP Status Code for "Reset Content"

HTTP_SEEOTHER = 303;
   HTTP Status Code for "See Other"

HTTP_SERVERERROR = 500;
   HTTP Status Code for "Internal Server Error"

HTTP_SUCCESS = 200;
   HTTP Status Code for "Success"

HTTP_SWITCHINGPROTOCOLS = 101;
   HTTP Status Code for "Switching Protocols"

HTTP_TEMPORARYREDIRECT = 307;
   HTTP Status Code for "Temporary Redirect"

HTTP_TIMEOUT = 408;
   HTTP Status Code for "Request Time-out"
HTTP_UNAUTHORIZED = 401;
    HTTP Status Code for "Unauthorized"

HTTP_UNAVAILABLE = 503;
    HTTP Status Code for "Service Unavailable"

HTTP_USEPROXY = 305;
    HTTP Status Code for "Use Proxy"

INITIALIZETABLE_NOINDEX: TSQLInitializeTableOptions =
    [itoNoIndex4ID..itoNoIndex4RecordVersion];
    Options to specify no index createon for TSQLRestServer.CreateMissingTables and
tSQLRecord.InitializeTable methods

INSERT_WITH_ID = [rFTS3, rFTS4, rFTS5, rRTree, rRTreeInteger, rCustomForcedID];
    If the TSQLRecordVirtual table kind expects the ID to be set on INSERT

IS_CUSTOM_VIRTUAL = [rCustomForcedID, rCustomAutoID];
    If the TSQLRecordVirtual table kind is not an embedded type
    - can be set for a TSQLRecord after a VirtualTableExternalRegister call

IS_FTS = [rFTS3, rFTS4, rFTS5];
    If the TSQLRecordVirtual table kind is a FTS virtual table

JSONSERIALIZEROPTIONS_AJAX = [jwoAsJsonNotAsString, jwoID_str];
    Typical TJSONSerializerSQLRecordOptions values for AJAX clients

JSONTOOBJECT_TOLERANTOPTIONS = [j2oHandleCustomVariants, j2oIgnoreUnknownEnum,
j2oIgnoreUnknownProperty, j2oIgnoreStringType, j2oAllowInt64Hex];
    Some open-minded options for JSONToObject() parsing
    - won't block JSON unserialization due to some minor class type definitions
    - used e.g. by TObjArraySerializer.CustomReader and
    TInterfacedObjectFake.FakeCall/TServiceMethodExecute.ExecuteJson methods

MAX_METHOD_ARGS = 32;
    Maximum number of method arguments handled by interfaces
    - if you consider this as a low value, you should better define some records/classes as DTOs instead
    of multiplying parameters: so don't ask to increase this value, we rather encourage writing clean
code
    - used e.g. to avoid creating dynamic arrays if not needed, and ease method calls

MAX_METHOD_COUNT = 128;
    Maximum number of methods handled by interfaces
    - if you think this constant is too low, you are about to break the "Interface Segregation" SOLID
    principle: so don't ask to increase this value, we won't allow to write un-SOLID code! :)

MAX_SQLTABLES = 256;
    Maximum number of Tables in a Database Model
    - this constant is used internally to optimize memory usage in the generated asm code
    - you should not change it to a value lower than expected in an existing database (e.g. as expected
    by TSQLAccessRights or such)

NORESPONSE_CONTENT_TYPE = '!NORESPONSE';
    Used to notify e.g. the THttpServerRequest not to wait for any response from the client
    - is not to be used in normal HTTP process, but may be used e.g. by
TWebSocketProtocolRest.ProcessFrame() to avoid to wait for an incoming response from the other endpoint
- should match HTTP_RESP_NORES c e n t constant defined in SynCrtSock.pas unit

NOT_SIMPLE_F I E L D S : TSQLFieldTypes = [sftUnknown, sftBlob, sftMany, sftRecordVersion];
Kind of fields not retrieved during normal query, update or adding
- by definition, BLOB are excluded to save transmission bandwidth
- by design, TSQLRecordMany properties are stored in an external pivot table
- by convenience, the TRecordVersion number is for internal use only

NO_DEFAULT = longint($80000000);
Back to normal alignment

NULLABLE_TYPES = [sftInteger, sftBoolean, sftEnumerate, sftFloat, sftCurrency, sftDateTime, sftTimeLog, sftUTF8Text];
The SQL field property types with their TNullable* equivalency
- those types may be stored in a variant published property, e.g.
  property Int: TNullableInteger read fInt write fInt;
  property Txt: TNullableUTF8Text read fTxt write fTxt;
  property Txt: TNullableUTF8Text index 32 read fTxt write fTxt;

PAGINGPARAMETERS_YAHOO: TSQLRestServerURIPagingParameters = ( Sort: 'SORT='; Dir: 'DIR='; StartIndex: 'STARTINDEX='; Results: 'RESULTS='; Select: 'SELECT='; Where: 'WHERE='; SendTotalRowsCountFmt: '');
The default URI parameters for query paging
- those values are the one expected by YUI components

RAWT E X T _ F I E L D S : TSQLFieldTypes = [sftAnsiText, sftUTF8Text];
Kind of fields which will contain pure TEXT values
- independently from the actual storage level
- i.e. will match RawUTF8, string, UnicodeString, WideString properties

RTREE_MAX_DIMENSION = 5;
Maximum handled dimension for TSQLRecordRTree
- this value is the one used by SQLite3 R-Tree virtual table

SERVERDEFAULTM O N I T O R L E V E L S : TSQLRestServerMonitorLevels = [mlTables, mlMethods, mlInterfaces, mlSQLite3];
Default value of TSQLRestServer.StatLevels property
- i.e. gather all statistics, but mlSessions

SERVICE.Contract_NONE_EXPECTED = '"';
Custom contract value to ignore contract validation from client side
- you could set the aContractExpected parameter to this value for TSQLRestClientURI.ServiceDefine or TSQLRestClientURI.ServiceRegister so that the contract won't be checked with the server
- it will be used e.g. if the remote server is not a mORMot server, but a plain REST/HTTP server - e.g. for public API notifications

SERVICE_IMPLEMENTATION_NOID = [sicSingle, sicShared];
The Server-side instance implementation patterns without any ID

STATICFILE_CONTENT_TYPE = '!STATICFILE';
Internal HTTP content-type for efficient static file sending
- detected e.g. by http.sys' THttpApiServer.Request or via the NGINX X-Accel-Redirect header's
THttpServer.Process for direct sending
- the OutCustomHeader should contain the proper 'Content-type: ....' corresponding to the file (e.g. by calling GetMimeContentType() function from SynCommons supplyings the file name)
- should match HTTP_RESP_STATICFILE constant defined in SynCrtSock.pas unit

```
STATICFILE_CONTENT_TYPE_HEADER = HEADER_CONTENT_TYPE+STATICFILE_CONTENT_TYPE;
```

Internal HTTP content-type Header for efficient static file sending

```
STATICFILE_CONTENT_TYPE_HEADER_UPPER = HEADER_CONTENT_TYPE_UPPER+STATICFILE_CONTENT_TYPE;
```

Uppercase version of HTTP header for static file content serving

```
STRING_FIELDS: TSQLFieldTypes = [sftAnsiText, sftUTF8Text, sftUTF8Custom, sftDateTime, sftDateTimeMS];
```

Kind of fields which will be stored as TEXT values
- i.e. RAWTEXT_FIELDS and TDateTime/TDateTimeMS

```
SUPERVISOR_ACCESS_RIGHTS: TSQLAccessRights = (AllowRemoteExecute: [reSQLSelectWithoutTable, reService, reUrlEncodedSQL, reUrlEncodedDelete]; GET: ALL_ACCESS_RIGHTS; POST: ALL_ACCESS_RIGHTS; PUT: ALL_ACCESS_RIGHTS; DELETE: ALL_ACCESS_RIGHTS);
```

Supervisor Database access right, i.e. allmighty over all Tables
- but WITHOUT the possibility to remotely execute any SQL statement (reSQL)
- is used as reference to create TSQLAuthUser 'Supervisor' access policy

```
SYNMONITORVALUE_CUMULATIVE = [smvMicroSec, smvBytes, smvCount, smvCount64];
```

Kind of "cumulative" TSynMonitorType stored in TSynMonitor / TSynMonitorUsage
- those properties will have their values reset for each granularity level
- will recognize TSynMonitorTotalMicroSec, TSynMonitorTotalBytes, TSynMonitorOneBytes, TSynMonitorBytesPerSec, TSynMonitorCount and TSynMonitorCount64 types

```
TEXT_DBFIELDS: TSQLDBFieldTypes = [ftUTF8, ftDate];
```

Kind of DB fields which will contain TEXT content when converted to JSON

```
VIRTUAL_TABLE_IGNORE_COLUMN = -2;
```

If a TSQLVirtualTablePreparedConstraint.Column is to be ignored

```
VIRTUAL_TABLE_ROWID_COLUMN = -1;
```

If a TSQLVirtualTablePreparedConstraint.Column points to the RowID

```
WM_TIMER_REFRESH_REPORT = 2;
```

Timer identifier which indicates we must refresh the Report content
- used for User Interface generation
- is handled in TSQLRibbon.RefreshClickHandled

```
WM_TIMER_REFRESH_SCREEN = 1;
```

Timer identifier which indicates we must refresh the current Page
- used for User Interface generation
- is associated with the TSQLRibbonTabParameters.AutoRefresh property, and is handled in TSQLRibbon.RefreshClickHandled

**Functions or procedures implemented in the mORMot unit**
<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddID</td>
<td>Similar to AddInt64() function, but for a TIDDynArray</td>
<td>2259</td>
</tr>
<tr>
<td>AddID</td>
<td>Similar to AddInt64() function, but for a TIDDynArray</td>
<td>2259</td>
</tr>
<tr>
<td>AdministrationExecute</td>
<td>May be used by DatabaseExecute/AdministrationExecute methods to serve a folder content for remote administration</td>
<td>2259</td>
</tr>
<tr>
<td>AuthURI</td>
<td>Computes an URI with optional jwt authentication parameter</td>
<td>2259</td>
</tr>
<tr>
<td>Base64MagicToBlob</td>
<td>Convert a Base64-encoded content into binary hexadecimal ready for SQL</td>
<td>2259</td>
</tr>
<tr>
<td>BlobToBytes</td>
<td>Create a TBytes from TEXT-encoded blob data</td>
<td>2259</td>
</tr>
<tr>
<td>BlobToStream</td>
<td>Create a memory stream from TEXT-encoded blob data</td>
<td>2259</td>
</tr>
<tr>
<td>BlobToTSQLRawBlob</td>
<td>Fill a TSQLRawBlob from TEXT-encoded blob data</td>
<td>2259</td>
</tr>
<tr>
<td>ClassFieldAllProps</td>
<td>Retrieve the PPropInfo values of all published properties of a class</td>
<td>2260</td>
</tr>
<tr>
<td>ClassFieldCountWithParents</td>
<td>Retrieve the total number of properties for a class, including its parents</td>
<td>2260</td>
</tr>
<tr>
<td>ClassFieldInstance</td>
<td>Retrieve a class Field property instance from a Property class type</td>
<td>2260</td>
</tr>
<tr>
<td>ClassFieldInstance</td>
<td>Retrieve a class Field property instance from a Property Name</td>
<td>2260</td>
</tr>
<tr>
<td>ClassFieldInstances</td>
<td>Retrieve all class Field property instances from a Property class type</td>
<td>2260</td>
</tr>
<tr>
<td>ClassFieldInt64</td>
<td>Retrieve an integer/Int64 Field property value from a Property Name</td>
<td>2260</td>
</tr>
<tr>
<td>ClassFieldNamesAllProps</td>
<td>Retrieve the field names of all published properties of a class</td>
<td>2260</td>
</tr>
<tr>
<td>ClassFieldNamesAllPropsAsText</td>
<td>Retrieve the field names of all published properties of a class</td>
<td>2260</td>
</tr>
<tr>
<td>ClassFieldProp</td>
<td>Retrieve a Field property RTTI information from a Property Name</td>
<td>2260</td>
</tr>
<tr>
<td>ClassFieldPropInstanceMatchingClass</td>
<td>Retrieve a class instance property value matching a class type</td>
<td>2261</td>
</tr>
<tr>
<td>ClassFieldPropWithParents</td>
<td>Retrieve a Field property RTTI information from a Property Name</td>
<td>2261</td>
</tr>
<tr>
<td>ClassFieldPropWithParentsFromClassOffset</td>
<td>Retrieve a Field property RTTI information searching for an exact Property offset address</td>
<td>2261</td>
</tr>
<tr>
<td>ClassFieldPropWithParentsFromClassType</td>
<td>Retrieve a Field property RTTI information searching for an exact Property class type</td>
<td>2261</td>
</tr>
<tr>
<td>ClassFieldPropWithParentsFromUTF8</td>
<td>Retrieve a Field property RTTI information from a Property Name</td>
<td>2261</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>ClassFieldPropWithParentsInheritsFromClass</td>
<td>Retrieve Field property RTTI information searching for an inherited Property class type</td>
<td>2261</td>
</tr>
<tr>
<td>ClassHasPublishedFields</td>
<td>Returns TRUE if the class has some published fields, including its parents</td>
<td>2261</td>
</tr>
<tr>
<td>ClassHierarchyWithField</td>
<td>Retrieve all class hierarchy types which have some published properties</td>
<td>2261</td>
</tr>
<tr>
<td>ClearObject</td>
<td>Will reset all the object properties to their default</td>
<td>2261</td>
</tr>
<tr>
<td>CopyCollection</td>
<td>Copy two TCollection instances</td>
<td>2261</td>
</tr>
<tr>
<td>CopyObject</td>
<td>Create a new object instance, from an existing one</td>
<td>2262</td>
</tr>
<tr>
<td>CopyObject</td>
<td>Copy object properties</td>
<td>2262</td>
</tr>
<tr>
<td>CopyStrings</td>
<td>Copy two TStrings instances</td>
<td>2262</td>
</tr>
<tr>
<td>CurrentServerNonce</td>
<td>Returns a safe 256-bit hexadecimal nonce, changing every 5 minutes</td>
<td>2262</td>
</tr>
<tr>
<td>CurrentServiceContext Server</td>
<td>Wrapper function to retrieve the current REST server instance from the global ServiceContext threadvar value</td>
<td>2262</td>
</tr>
<tr>
<td>DocVariantToObject</td>
<td>Fill a class instance from a TDocVariant object document properties</td>
<td>2262</td>
</tr>
<tr>
<td>GetEnumCaption</td>
<td>Retrieve the ready to be displayed text of an enumeration</td>
<td>2262</td>
</tr>
<tr>
<td>GetEnumNameTrimed</td>
<td>Get the corresponding enumeration name, without the first lowercase chars (otDone -&gt; 'Done')</td>
<td>2263</td>
</tr>
<tr>
<td>GetInterfaceFromEntry</td>
<td>Execute an instance method from its RTTI per-interface information</td>
<td>2263</td>
</tr>
<tr>
<td>GetJSONObjectAsSQL</td>
<td>Decode JSON fields object into an UTF-8 encoded SQL-ready statement</td>
<td>2263</td>
</tr>
<tr>
<td>GetObjectNameCSV</td>
<td>Get all included values of an enumeration set, as CSV names</td>
<td>2263</td>
</tr>
<tr>
<td>GetTableNamesFromSQLSelect</td>
<td>Naive search of '... FROM TableName ...' pattern in the supplied SQL</td>
<td>2264</td>
</tr>
<tr>
<td>GetTableNamesFromSQLSelect</td>
<td>Naive search of '... FROM Table1,Table2 ...' pattern in the supplied SQL</td>
<td>2264</td>
</tr>
<tr>
<td>InterfaceArrayDeleteAfterException</td>
<td>Safe deletion of a T*InterfaceArray dynamic array item</td>
<td>2264</td>
</tr>
<tr>
<td>InternalClassProp</td>
<td>Retrieve the class property RTTI information for a specific class</td>
<td>2264</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>InternalClassPropInfo</td>
<td>Retrieve the class property RTTI information for a specific class</td>
<td>2264</td>
</tr>
<tr>
<td>InternalMethodInfo</td>
<td>Retrieve a method RTTI information for a specific class</td>
<td>2264</td>
</tr>
<tr>
<td>IsBlobHex</td>
<td>Return true if the TEXT is encoded as SQLite3 BLOB literals (X'53514C697465' e.g.)</td>
<td>2264</td>
</tr>
<tr>
<td>IsValidHttpHeader</td>
<td>Check the supplied HTTP header to not contain more than one EOL</td>
<td>2264</td>
</tr>
<tr>
<td>IsNotAjaxJSON</td>
<td>Returns TRUE if the JSON content is in expanded format</td>
<td>2264</td>
</tr>
<tr>
<td>IsObjectDefaultOrVoid</td>
<td>Returns TRUE on a nil instance or if all its published properties are default/0</td>
<td>2264</td>
</tr>
<tr>
<td>JSONFileToObject</td>
<td>Fill the object properties from a JSON file content</td>
<td>2265</td>
</tr>
<tr>
<td>JSONGetID</td>
<td>Retrieve the ID/RowID field of a JSON object</td>
<td>2265</td>
</tr>
<tr>
<td>JSONGetObject</td>
<td>Retrieve a JSON '{&quot;Name&quot;:Value,....}' object</td>
<td>2265</td>
</tr>
<tr>
<td>JSONSettingsToObject</td>
<td>Parse the supplied JSON with some tolerance about Settings format</td>
<td>2265</td>
</tr>
<tr>
<td>JSONToNewObject</td>
<td>Create a new object instance, as saved by ObjectToJSON(...,[...],woStoreClassName,...);</td>
<td>2265</td>
</tr>
<tr>
<td>JSONToObject</td>
<td>Read an object properties, as saved by ObjectToJSON function</td>
<td>2266</td>
</tr>
<tr>
<td>MonitorPropUsageValue</td>
<td>Guess the kind of value stored in a TSynMonitor / TsynMonitorUsage property</td>
<td>2266</td>
</tr>
<tr>
<td>ObjArrayCopy</td>
<td>Wrapper to create a new T<em>ObjArray with copied instances of a source T</em>ObjArray</td>
<td>2266</td>
</tr>
<tr>
<td>ObjArrayRecordIDs</td>
<td>Wrapper to return all TID values of an array of TSQLRecord</td>
<td>2266</td>
</tr>
<tr>
<td>ObjArraySearch</td>
<td>Wrapper to search for a given TSQLRecord by ID in an array of TSQLRecord</td>
<td>2266</td>
</tr>
<tr>
<td>ObjectDefaultToVariant</td>
<td>Will convert a blank TObject into a TDocVariant document instance</td>
<td>2266</td>
</tr>
<tr>
<td>ObjectEquals</td>
<td>Is able to compare two objects by value</td>
<td>2267</td>
</tr>
<tr>
<td>ObjectFromInterface</td>
<td>Low-level function to retrieve the class instance implementing a given interface</td>
<td>2267</td>
</tr>
<tr>
<td>ObjectFromInterfaceImpleme nts</td>
<td>Low-level function to check if a class instance, retrieved from its interface variable, does in fact implement a given interface</td>
<td>2267</td>
</tr>
<tr>
<td>ObjectLoadJSON</td>
<td>Read an object properties, as saved by ObjectToJSON function</td>
<td>2267</td>
</tr>
<tr>
<td>ObjectLoadVariant</td>
<td>Read an object properties from a TDocVariant object document</td>
<td>2267</td>
</tr>
<tr>
<td>ObjectToJSONDebug</td>
<td>Will serialize any TObject into its expanded UTF-8 JSON representation</td>
<td>2267</td>
</tr>
<tr>
<td>ObjectToJSONFile</td>
<td>Persist a class instance into a JSON file</td>
<td>2267</td>
</tr>
<tr>
<td><strong>Functions or procedures</strong></td>
<td><strong>Description</strong></td>
<td><strong>Page</strong></td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------</td>
<td>---------</td>
</tr>
<tr>
<td><code>ObjectToVariantDebug</code></td>
<td>Will serialize any TObject into a TDocVariant debugging document</td>
<td>2268</td>
</tr>
<tr>
<td><code>ObjectToVariantDebug</code></td>
<td>Will serialize any TObject into a TDocVariant debugging document</td>
<td>2268</td>
</tr>
<tr>
<td><code>ReadObject</code></td>
<td>Read an object properties, as saved by TINIWriter.WriteObject() method</td>
<td>2268</td>
</tr>
<tr>
<td><code>ReadObject</code></td>
<td>Read an object properties, as saved by TINIWriter.WriteObject() method</td>
<td>2268</td>
</tr>
<tr>
<td><code>RecordReference</code></td>
<td>Create a TRecordReference with the corresponding parameters</td>
<td>2268</td>
</tr>
<tr>
<td><code>RecordReference</code></td>
<td>Create a TRecordReference with the corresponding parameters</td>
<td>2268</td>
</tr>
<tr>
<td><code>RecordRefToID</code></td>
<td>Convert a dynamic array of TRecordReference into its corresponding IDs</td>
<td>2268</td>
</tr>
<tr>
<td><code>SelectInClause</code></td>
<td>Compute 'PropName in (...)’ where clause for a SQL statement</td>
<td>2269</td>
</tr>
<tr>
<td><code>SelectInClause</code></td>
<td>Compute 'PropName in (...)’ where clause for a SQL statement</td>
<td>2269</td>
</tr>
<tr>
<td><code>SetDefaultValuesObject</code></td>
<td>Set any default integer or enumerates (including boolean) published properties values for a TPersistent/TSynPersistent</td>
<td>2269</td>
</tr>
<tr>
<td><code>SetID</code></td>
<td>Set the TID (=64-bit integer) value from the numerical text stored in P^</td>
<td>2269</td>
</tr>
<tr>
<td><code>SetID</code></td>
<td>Set the TID (=64-bit integer) value from the numerical text stored in U</td>
<td>2269</td>
</tr>
<tr>
<td><code>SetWeak</code></td>
<td>Assign a Weak interface reference, to be used for circular references</td>
<td>2269</td>
</tr>
<tr>
<td><code>SetWeakZero</code></td>
<td>{$ifdef HASINLINE}inline;{$endif} raise compilation Internal Error C2170 assign a Weak interface reference, which will be ZEROed (set to nil) when the corresponding object will be released</td>
<td>2270</td>
</tr>
<tr>
<td><code>SQLFromWhere</code></td>
<td>Compute the SQL corresponding to a WHERE clause</td>
<td>2270</td>
</tr>
<tr>
<td><code>SQLGetOrder</code></td>
<td>Get the order table name from a SQL statement</td>
<td>2270</td>
</tr>
<tr>
<td><code>SQLWhereIsEndClause</code></td>
<td>Find out if the supplied WHERE clause starts with one of the ORDER/GROUP/LIMIT/OFFSET/JOIN keywords</td>
<td>2270</td>
</tr>
<tr>
<td><code>StatusCodeIsSuccess</code></td>
<td>Returns true for successful HTTP status codes, i.e. in 200..399 range</td>
<td>2270</td>
</tr>
<tr>
<td><code>StatusCodeToErrorMsg</code></td>
<td>Convert any HTTP_* constant to an integer error code and its English text</td>
<td>2270</td>
</tr>
<tr>
<td><code>ToMethod</code></td>
<td>Convert a string HTTP verb into its TSQLURIMethod enumerate</td>
<td>2270</td>
</tr>
<tr>
<td><code>ToText</code></td>
<td>Returns the interface name of a registered GUID, or its hexadecimal value</td>
<td>2270</td>
</tr>
<tr>
<td><code>TSQLRawBlobToBlob</code></td>
<td>Creates a TEXT-encoded version of blob data from a memory data</td>
<td>2271</td>
</tr>
<tr>
<td><code>TSQLRawBlobToBlob</code></td>
<td>Creates a TEXT-encoded version of blob data from a TSQLRawBlob</td>
<td>2271</td>
</tr>
<tr>
<td><code>TSQLRecordDynArrayCompare</code></td>
<td>TDynArraySortCompare compatible function, sorting by TSQLRecord.ID</td>
<td>2271</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>TSQLRecordDynArrayHashOne</td>
<td>TDynArrayHashOne compatible function, hashing TSQLRecord.ID</td>
<td>2271</td>
</tr>
<tr>
<td>UnJSONFirstField</td>
<td>Get the FIRST field value of the FIRST row, from a JSON content</td>
<td>2271</td>
</tr>
<tr>
<td>URIRequest</td>
<td>This function can be exported from a DLL to remotely access to a TSQLRestServer</td>
<td>2271</td>
</tr>
<tr>
<td>UriDecodeObject</td>
<td>Decode a specified parameter compatible with URI encoding into its original object contents</td>
<td>2271</td>
</tr>
<tr>
<td>UTF8CompareBoolean</td>
<td>Special comparison function for sorting sftBoolean UTF-8 encoded values in the SQLite3 database or JSON content</td>
<td>2271</td>
</tr>
<tr>
<td>UTF8CompareCurr64</td>
<td>Special comparison function for sorting sftCurrency UTF-8 encoded values in the SQLite3 database or JSON content</td>
<td>2271</td>
</tr>
<tr>
<td>UTF8CompareDouble</td>
<td>Special comparison function for sorting sftFloat UTF-8 encoded values in the SQLite3 database or JSON content</td>
<td>2271</td>
</tr>
<tr>
<td>UTF8CompareInt64</td>
<td>Special comparison function for sorting sftInteger, sftTID, sftRecordVersion sftTimeLog/sftModTime/sftCreateTime or sftUnixTime/sftUnixMSTime UTF-8 encoded values in the SQLite3 database or JSON content</td>
<td>2272</td>
</tr>
<tr>
<td>UTF8CompareISO8601</td>
<td>Special comparison function for sorting sftDateTime or sftDateTimeMS UTF-8 encoded values in the SQLite3 database or JSON content</td>
<td>2272</td>
</tr>
<tr>
<td>UTF8CompareRecord</td>
<td>Special comparison function for sorting TRecord (TRecordReference/RecordRef) UTF-8 encoded values in the SQLite3 database or JSON content</td>
<td>2272</td>
</tr>
<tr>
<td>UTF8CompareUInt32</td>
<td>Special comparison function for sorting sftEnumerate, sftSet or sftID UTF-8 encoded values in the SQLite3 database or JSON content</td>
<td>2272</td>
</tr>
<tr>
<td>UTF8ContentNumberType</td>
<td>Guess the number type of an UTF-8 encoded field value, as used in TSQLTable.Get()</td>
<td>2272</td>
</tr>
<tr>
<td>UTF8ContentType</td>
<td>Guess the content type of an UTF-8 encoded field value, as used in TSQLTable.Get()</td>
<td>2272</td>
</tr>
<tr>
<td>ValueVarToVariant</td>
<td>Low-level function used to convert a JSON Value into a variant, according to the property type</td>
<td>2272</td>
</tr>
<tr>
<td>WriteObject</td>
<td>Write an object properties, as saved by TINIWriter.WriteObject() method</td>
<td>2273</td>
</tr>
<tr>
<td>WriteObject</td>
<td>Write an object properties, as saved by TINIWriter.WriteObject() method</td>
<td>2273</td>
</tr>
<tr>
<td>_ObjAddProps</td>
<td>Add the property values of a TObject to a document-based object content</td>
<td>2273</td>
</tr>
</tbody>
</table>
**procedure** AddID(var Values: TIDDynArray; Value: TID); overload;

*Similar to AddInt64() function, but for a TIDDynArray*
- some random GPF were identified with AddInt64(TInt64DynArray(Values),...) with the Delphi Win64 compiler

**procedure** AddID(var Values: TIDDynArray; var ValuesCount: integer; Value: TID); overload;

*Similar to AddInt64() function, but for a TIDDynArray*
- some random GPF were identified with AddInt64(TInt64DynArray(Values),...) with the Delphi Win64 compiler

**procedure** AdministrationExecuteGetFiles(const Folder, Mask: TFileName; const Param: RawUTF8; var Answer: TServiceCustomAnswer);

*May be used by DatabaseExecute/AdministrationExecute methods to serve a folder content for remote administration*

**function** AuthURI(const URI, URIAuthenticationBearer: RawUTF8): RawUTF8;

*Computes an URI with optional jwt authentication parameter*
- if AuthenticationBearer is set, will add its values as additional parameter:
  URI?authentication bearer=URIAuthenticationBearer

**procedure** Base64MagicToBlob(Base64: PUTF8Char; var result: RawUTF8);

*Convert a Base64-encoded content into binary hexadecimal ready for SQL*
- returns e.g. X'53514C697465'

**function** BlobToBytes(P: PUTF8Char): TBytes;

*Create a TBytes from TEXT-encoded blob data*
- blob data can be encoded as SQLite3 BLOB literals (X'53514C697465' e.g.) or Base-64 encoded content ('\uFFF0base64encodedbinary') or plain TEXT

**function** BlobToStream(P: PUTF8Char): TStream;

*Create a memory stream from TEXT-encoded blob data*
- blob data can be encoded as SQLite3 BLOB literals (X'53514C697465' e.g.) or Base-64 encoded content ('\uFFF0base64encodedbinary') or plain TEXT
- the caller must free the stream instance after use

**procedure** BlobToTSQLRawBlob(P: PUTF8Char; var result: TSQLRawBlob); overload;

*Fill a TSQLRawBlob from TEXT-encoded blob data*
- blob data can be encoded as SQLite3 BLOB literals (X'53514C697465' e.g.) or Base-64 encoded content ('\uFFF0base64encodedbinary') or plain TEXT

**procedure** BlobToTSQLRawBlob(const Blob: RawByteString): TSQLRawBlob; overload;

*Fill a TSQLRawBlob from TEXT-encoded blob data*
- blob data can be encoded as SQLite3 BLOB literals (X'53514C697465' e.g.) or Base-64 encoded content ('\uFFF0base64encodedbinary') or plain TEXT

**function** BlobToTSQLRawBlob(P: PUTF8Char): TSQLRawBlob; overload;

*Fill a TSQLRawBlob from TEXT-encoded blob data*
- blob data can be encoded as SQLite3 BLOB literals (X'53514C697465' e.g.) or Base-64 encoded content ('\uFFF0base64encodedbinary') or plain TEXT
function ClassFieldAllProps(ClassType: TClass; Types: TTypeKinds=[low(TTypeKind)..high(TTypeKind)]): PPropInfoDynArray;

    Retrieve the PPropInfo values of all published properties of a class
    - you could select which property types should be included in the list

function ClassFieldCountWithParents(ClassType: TClass; onlyWithoutGetter: boolean=false): integer;

    Retrieve the total number of properties for a class, including its parents

function ClassFieldInstance(Instance: TObject; const PropName: shortstring; PropClassType: TClass; out PropInstance): boolean; overload;

    Retrieve a class Field property instance from a Property Name
    - this version also search into parent properties
    - returns TRUE and set PropInstance if a matching property was found

function ClassFieldInstance(Instance: TObject; PropClassType: TClass; out PropInstance): boolean; overload;

    Retrieve a class Field property instance from a Property class type
    - this version also search into parent properties
    - returns TRUE and set PropInstance if a matching property was found

function ClassFieldInstances(Instance: TObject; PropClassType: TClass): TObjectDynArray;

    Retrieve all class Field property instances from a Property class type
    - this version also search into parent properties
    - returns all matching property instances found

function ClassFieldInt64(Instance: TObject; const PropName: ShortString; out PropValue: Int64): boolean;

    Retrieve an integer/Int64 Field property value from a Property Name
    - this version also search into parent properties
    - returns TRUE and set PropValue if a matching property was found

function ClassFieldNameAllProps(ClassType: TClass; IncludePropType: boolean=false; Types: TTypeKinds=[low(TTypeKind)..high(TTypeKind)]): TRawUTF8DynArray;

    Retrieve the field names of all published properties of a class
    - will optionally append the property type to the name, e.g 'Age: integer'
    - you could select which property types should be included in the list

function ClassFieldNameAllPropsAsText(ClassType: TClass; IncludePropType: boolean=false; Types: TTypeKinds=[low(TTypeKind)..high(TTypeKind)]): RawUTF8;

    Retrieve the field names of all published properties of a class
    - will optionally append the property type to the name, e.g 'Age: integer'
    - you could select which property types should be included in the list

function ClassFieldProp(ClassType: TClass; const PropName: shortstring): PPropInfo;

    Retrieve a Field property RTTI information from a Property Name
function ClassFieldPropInstanceMatchingClass(aSearchedInstance: TObject; aSearchedClassType: TClass): TObject;
    Retrieve a class instance property value matching a class type
    - if aSearchedInstance is aSearchedClassType, will return aSearchedInstance
    - if aSearchedInstance is not aSearchedClassType, it will try all nested properties of
      aSearchedInstance for a matching aSearchedClassType: if no exact match is found, will return
      aSearchedInstance

function ClassFieldPropWithParents(aClassType: TClass; const aPropName: shortstring; aCaseSensitive: boolean=false): PPropInfo;
    Retrieve a Field property RTTI information from a Property Name
    - this special version also search into parent properties (default is only current)

function ClassFieldPropWithParentsFromClassOffset(aClassType: TClass; aSearchedOffset: pointer): PPropInfo;
    Retrieve a Field property RTTI information searching for an exact Property offset address
    - this special version also search into parent properties

function ClassFieldPropWithParentsFromClassType(aClassType,aSearchedClassType: TClass): PPropInfo;
    Retrieve a Field property RTTI information searching for an exact Property class type
    - this special version also search into parent properties

function ClassFieldPropWithParentsFromClassUTF8(aClassType: TClass; PropName: PUTF8Char; PropNameLen: integer; aCaseSensitive: boolean=false): PPropInfo;
    Retrieve a Field property RTTI information from a Property Name
    - this special version also search into parent properties (default is only current)

function ClassFieldPropWithParentsInheritsFromClassType(aClassType,aSearchedClassType: TClass): PPropInfo;
    Retrieve a Field property RTTI information searching for an inherited Property class type
    - this special version also search into parent properties

function ClassHasPublishedFields(ClassType: TClass): boolean;
    Returns TRUE if the class has some published fields, including its parents

function ClassHierarchyWithField(ClassType: TClass): TClassDynArray;
    Retrieve all class hierachy types which have some published properties

procedure ClearObject(Value: TObject; FreeAndNilNestedObjects: boolean=false);
    Will reset all the object properties to their default
    - strings will be set to ", numbers to 0
    - if FreeAndNilNestedObjects is the default FALSE, will recursively reset all nested class properties
    - if FreeAndNilNestedObjects is TRUE, will FreeAndNil() all the nested class properties
    - for a TSQLRecord, use its ClearProperties method instead, which will handle the ID property, and
      any nested JOINed instances

procedure CopyCollection(Source, Dest: TCollection);
    Copy two TCollection instances
    - will call CopyObject() in loop to repopulate the Dest collection, which will work even if Assign() method was not overriden
procedure CopyObject(aFrom, aTo: TObject); overload;

*Copy object properties*
- copy Integer, Int64, enumerates (including boolean), variant, records, dynamic arrays, classes and any string properties (excluding shortstring)
- TCollection items can be copied also, if they are of the same exact class
- object properties instances are created in aTo if the objects are not TSQLRecord children (in this case, these are not class instances, but INTEGER reference to records, so only the integer value is copied), that is for regular Delphi classes

function CopyObject(aFrom: TObject): TObject; overload;

*Create a new object instance, from an existing one*
- will create a new instance of the same class, then call the overloaded CopyObject() procedure to copy its values

procedure CopyStrings(Source, Dest: TStrings);

*Copy two TStrings instances*
- will just call Dest.Assign(Source) in practice

function CurrentServerNonce(Previous: boolean=false): RawUTF8;

*Returns a safe 256-bit hexadecimal nonce, changing every 5 minutes*
- as used e.g. by TSQLRestServerAuthenticationDefault.Auth
- this function is very fast, even if cryptographically-level SHA-3 secure

function CurrentServiceContext: TServiceRunningContext;

*Wrapper function to retrieve the global ServiceContext threadvar value*
- to be used when accessing the value from a package, to circumvent a Delphi RTL/compiler restriction (bug?)
- for a cleaner SOA/DI approach, consider using TInjectableObjectRest

function CurrentServiceContextServer: TSQLRestServer;

*Wrapper function to retrieve the current REST server instance from the global ServiceContext threadvar value*
- may return nil if ServiceContext.Request is nil: in this case, you should better implement your service by inheriting the implementation class from TInjectableObjectRest

procedure DocVariantToObject(var doc: TDocVariantData; obj: TObject): boolean;

*Fill a class instance from a TDocVariant object document properties*
- returns FALSE if the variant is not a dvObject, TRUE otherwise

function GetEnumCaption(aTypeInfo: PTypeInfo; const aIndex): string;

*Retrieve the ready to be displayed text of an enumeration*
- will "uncamel" then translate into a generic VCL string
- aIndex will be converted to the matching ordinal value (byte or word)
function GetEnumNameTrimed(aTypeInfo: PTypeInfo; const aIndex): RawUTF8;

Get the corresponding enumeration name, without the first lowercase chars (otDone -> 'Done')
- aIndex will be converted to the matching ordinal value (byte or word)
- this will return the code-based English text; use GetEnumCaption() to retrieve the enumeration display text

function GetInterfaceFromEntry(Instance: TObject; Entry: PInterfaceEntry; out Obj): boolean;

Execute an instance method from its RTTI per-interface information
- calling this function with a pre-computed PInterfaceEntry value is faster than calling the TObject.GetInterface() method, especially when the class implements several interfaces, since it avoid a slow GUID lookup

function GetJSONObjectAsSQL(var P: PUTF8Char; const Fields: TRawUTF8DynArray; Update, InlinedParams: boolean; RowID: TID=0; ReplaceRowIDWithID: Boolean=false): RawUTF8; overload;

Decode JSON fields object into an UTF-8 encoded SQL-ready statement
- this function decodes in the P^ buffer memory itself (no memory allocation or copy), for faster process - so take care that it is an unique string
- P should be after the initial '{' or '[' character, i.e. at first field
- P contains the next object start or nil on unexpected end of input
- if Fields is void, expects expanded "COL1"="VAL1" pairs in P^, stopping at '}' or ']'  
- otherwise, Fields[] contains the column names and expects "VAL1","VAL2".. in P^ 
- returns '{COL1="VAL1", COL2=VAL2' if UPDATE is true (UPDATE SET format)
- returns '(COL1, COL2) VALUES("VAL1", VAL2)' otherwise (INSERT format)
- escape SQL strings, according to the official SQLite3 documentation (i.e. ' inside a string is stored as '')
- if InlinedParams is set, will create prepared parameters like 'COL1=:\"VAL1\":, COL2=\:VAL2:\'
- if RowID is set, a RowID column will be added within the returned content

Used for DI-2.1.2 (page 2545).

function GetJSONObjectAsSQL(const JSON: RawUTF8; Update, InlinedParams: boolean; RowID: TID=0; ReplaceRowIDWithID: Boolean=false): RawUTF8; overload;

Decode JSON fields object into an UTF-8 encoded SQL-ready statement
- this function decodes in the P^ buffer memory itself (no memory allocation or copy), for faster process - so take care that it is an unique string
- P should be after the initial '{' or '[' character, i.e. at first field
- P contains the next object start or nil on unexpected end of input
- if Fields is void, expects expanded "COL1"="VAL1" pairs in P^, stopping at '}' or ']'  
- otherwise, Fields[] contains the column names and expects "VAL1","VAL2".. in P^ 
- returns '{COL1="VAL1", COL2=VAL2' if UPDATE is true (UPDATE SET format)
- returns '(COL1, COL2) VALUES("VAL1", VAL2)' otherwise (INSERT format)
- escape SQL strings, according to the official SQLite3 documentation (i.e. ' inside a string is stored as '')
- if InlinedParams is set, will create prepared parameters like 'COL1=:\"VAL1\":, COL2=\:VAL2:\'
- if RowID is set, a RowID column will be added within the returned content

Used for DI-2.1.2 (page 2545).

function GetObjectComponent(Obj: TPersistent; const ComponentName: shortstring; ComponentClass: TClass): pointer;

Retrieve an object's component from its property name and class
- useful to set User Interface component, e.g.

function GetSetNameCSV(aTypeInfo: PTypeInfo; const aValue): RawUTF8;

Get all included values of an enumeration set, as CSV names
function GetTableNameFromSQLSelect(const SQL: RawUTF8; EnsureUniqueTableInFrom: boolean): RawUTF8;
Naive search of '... FROM TableName ...' pattern in the supplied SQL

function GetTableNamesFromSQLSelect(const SQL: RawUTF8): TRawUTF8DynArray;
Naive search of '... FROM Table1,Table2 ...' pattern in the supplied SQL

procedure InterfaceArrayDeleteAfterException(var aInterfaceArray; const aItemIndex: integer; aLog: TSynLogFamily; const aLogMsg: RawUTF8; aInstance: TObject);
Safe deletion of a T*InterfaceArray dynamic array item
- similar to InterfaceArrayDelete, but with a safe try .. except block during the entry deletion (since the system may be unstable)
- will also log a warning with the Interface name (from aLogMsg) and aInstance

function InternalClassProp(ClassType: TClass): PClassProp;
Retrieve the class property RTTI information for a specific class

function InternalClassPropInfo(ClassType: TClass; out PropInfo: PPropInfo): integer;
Retrieve the class property RTTI information for a specific class
- will return the number of published properties
- and set the PropInfo variable to point to the first property
- typical use to enumerate all published properties could be:

```pascal
var i: integer;
CT: TClass;
P: PPropInfo;
begin
  CT := ..;
  repeat
    for i := 1 to InternalClassPropInfo(CT,P) do begin
      // use P^  
      P := P^.Next;
    end;
    CT := GetClassParent(CT);
  until CT=nil;
end;
```
such a loop is much faster than using the RTL's TypeInfo or RTTI units

function InternalMethodInfo(aClassType: TClass; const aMethodName: ShortString): PMethodInfo;
Retrieve a method RTTI information for a specific class

function IsBlobHex(P: PUTF8Char): boolean;
Return true if the TEXT is encoded as SQLite3 BLOB literals (X'53514C697465' e.g.)

function IsInvalidHTTPHeader(head: PUTF8Char; headlen: PtrInt): boolean;
Check the supplied HTTP header to not contain more than one EOL
- to avoid unexpected HTTP body injection, e.g. from unsafe business code

function IsNotAjaxJSON(P: PUTF8Char): Boolean;
Returns TRUE if the JSON content is in expanded format
- i.e. as plain ["ID":10,"FirstName":"John","LastName":"Smith"]...
- i.e. not as {"fieldCount":3,"values":{"ID","FirstName","LastName",...}"

function IsObjectDefaultOrVoid(Value: TObject): boolean;
Returns TRUE on a nil instance or if all its published properties are default/0
- calls internally TPropInfo.IsDefaultOrVoid()
function JSONFileToObject(const JSONFile: TFileName; var ObjectInstance; TObjectListItemClass: TClass=nil; Options: TJSONToObjectOptions=[]): boolean;

Fill the object properties from a JSON file content
- ObjectInstance must be an existing TObject instance
- this function will call RemoveCommentsFromJSON() before process

function JSONGetID(P: PUTF8Char; out ID: TID): Boolean;

Retrieve the ID/RowID field of a JSON object
- this function expects this "ID" property to be the FIRST in the "Name":Value pairs, as generated by TSQLRecord.GetJSONValues(W)
- returns TRUE if a ID/RowID>0 has been found, and set ID with the value

function JSONGetObject(var P: PUTF8Char; ExtractID: PID; var EndOfObject: AnsiChar; KeepIDField: boolean): RawUTF8;

Retrieve a JSON '{"Name":Value,...}' object
- P is nil in return in case of an invalid object
- returns the UTF-8 encoded JSON object, including first '{' and last '}
- if ExtractID is set, it will contain the "ID":203 field value, and this field won't be included in the resulting UTF-8 encoded JSON object unless KeepIDField is true
- this function expects this "ID" property to be the FIRST in the "Name":Value pairs, as generated by TSQLRecord.GetJSONValues(W)

function JSONSettingsToObject(var InitialJsonContent: RawUTF8; Instance: TObject): boolean;

Parse the supplied JSON with some tolerance about Settings format
- will make a TSynTempBuffer copy for parsing, and un-comment it
- returns true if the supplied JSON was successfully retrieved
- returns false and set InitialJsonContent := "" on error

function JSONToObject(var From: PUTF8Char; var Valid: boolean; Options: TJSONToObjectOptions=[]): TObject;

Create a new object instance, as saved by ObjectToJSON(...,[...,woStoreClassName,...]);
- JSON input should be either 'null', either '{"ClassName":"TMyClass","...}"
- woStoreClassName option shall have been used at ObjectToJSON() call
- and the corresponding class shall have been previously registered by TJSONSerializer.RegisterClassForJSON(), in order to retrieve the class type from it name - or, at least, by a Classes.RegisterClass() function call
- the data inside From^ is modified in-place (unescape and transformed): don't call JSONToObject(pointer(JSONRawUTF8)) but makes a temporary copy of the JSON text buffer before calling this function, if want to reuse it later
function JSONToObject(var ObjectInstance; From: PUTF8Char; out Valid: boolean; TObjectListItemClass: TClass=nil; Options: TJSONToObjectOptions=[]): PUTF8Char;

Read an object properties, as saved by ObjectToJSON function
- ObjectInstance must be an existing TObject instance
- the data inside From^ is modified in-place (unescape and transformed): calling
JSONToObject(pointer(JSONRawUTF8)) will change the JSONRawUTF8 variable content, which
may not be what you expect - consider using the ObjectLoadJSON() function instead
- handle Integer, Int64, enumerate (including boolean), set, floating point, TDateTime, TCollection,
TStrings, TRawUTF8List, variant, and string properties (excluding ShortString, but including
WideString and UnicodeString under Delphi 2009+)
- TList won't be handled since it may leak memory when calling TList.Clear
- won't handle TObjectList (even if ObjectToJSON is able to serialize them) since has no way of
knowing the object type to add (TCollection.Add is missing), unless: 1. you set the
TObjectListItemClass property as expected, and provide a TObjectList object, or 2.
woStoreClassName option has been used at ObjectToJSON() call and the corresponding classes
have been previously registered by TJSONSerializer.RegisterClassForJSON() (or
Classes.RegisterClass)
- will clear any previous TCollection objects, and convert any null JSON basic type into nil - e.g. if
From='null', will call FreeAndNil(Value)
- you can add some custom (un)serializers for ANY Delphi class, via the
TJSONSerializer.RegisterCustomSerializer() class method
- set Valid=TRUE on success, Valid=FALSE on error, and the main function will point in From at the
syntax error place (e.g. on any unknown property name)
- caller should explicitly perform a SetDefaultValuesObject(Value) if the default values are
expected to be set before JSON parsing

function MonitorPropUsageValue(info: PPropInfo): TSynMonitorType;

Guess the kind of value stored in a TSynMonitor / TSynMonitorUsage property
- will recognize TSynMonitorTotalMicroSec, TSynMonitorOneMicroSec, TSynMonitorTotalBytes,
TSynMonitorOneBytes, TSynMonitorBytesPerSec, TSynMonitorCount and TSynMonitorCount64
types from supplied RTTI

procedure ObjArrayCopy(const aSourceObjArray; var aDestObjArray; aDestObjArrayClear: boolean=true);

Wrapper to create a new T*ObjArray with copied instances of a source T*ObjArray
- use internally CopyObject() over aSourceObjArray[] instances
- will clear aDestObjArray before items copy, if aDestObjArrayClear = TRUE

procedure ObjArrayRecordIDs(const aSQLRecordObjArray; out result: TInt64DynArray);

Wrapper to return all TID values of an array of TSQLRecord

function ObjArraySearch(const aSQLRecordObjArray; aID: TID): TSQLRecord;

Wrapper to search for a given TSQLRecord by ID in an array of TSQLRecord

function ObjectDefaultToVariant(aClass: TClass; aOptions: TDocVariantOptions): TDocVariant;

Will convert a blank TObject into a TDocVariant document instance
**function** ObjectEquals(Value1, Value2: TObject; ignoreGetterFields: boolean=false): boolean;

*Is able to compare two objects by value*
- both instances may or may not be of the same class, but properties should match
- will use direct RTTI access of property values, or TSQLRecord.SameValues() if available to make the comparison as fast and accurate as possible
- if you want only to compare the plain fields with no getter function, e.g. if they are just some conversion of the same information, you can set ignoreGetterFields=TRUE

**function** ObjectFromInterface(const aValue: IInterface): TObject;

*Low-level function to retrieve the class instance implementing a given interface*
- this will work with interfaces stubs generated by the compiler, but also with TInterfaceFactory.CreateFakeInstance kind of classes
- returns nil if aValue is nil or not recognized

**function** ObjectFromInterfaceImplements(const aValue: IInterface; const aInterface: TGUID): boolean;

*Low-level function to check if a class instance, retrieved from its interface variable, does in fact implement a given interface*
- this will call ObjectFromInterface(), so will work with interfaces stubs generated by the compiler, but also with TInterfaceFactory.CreateFakeInstance kind of classes

**function** ObjectLoadJSON(var ObjectInstance; const JSON: RawUTF8; TObjectListItemClass: TClass=nil; Options: TJSONToObjectOptions=[]): boolean;

*Read an object properties, as saved by ObjectToJSON function*
- ObjectInstance must be an existing TObject instance
- this overloaded version will make a private copy of the supplied JSON content (via TSynTempBuffer), to ensure the original buffer won't be modified during process, before calling safely JSONToObject()
- will return TRUE on success, or FALSE if the supplied JSON was invalid

**function** ObjectLoadVariant(var ObjectInstance; const aDocVariant: variant; TObjectListItemClass: TClass=nil; Options: TJSONToObjectOptions=[]): boolean;

*Read an object properties from a TDocVariant object document*
- ObjectInstance must be an existing TObject instance
- will return TRUE on success, or FALSE if the supplied aDocVariant was not a TDocVariant object

**function** ObjectToJSONDebug(Value: TObject; Options: TTextWriterWriteObjectOptions=[woDontStoreDefault, woHumanReadable, woStoreClassName, woStorePointer]): RawUTF8;

*Will serialize any TObject into its expanded UTF-8 JSON representation*
- includes debugger-friendly information, similar to TSynLog, i.e. class name and sets/enumerates as text
- could be used to create a TDocVariant object with full information
- wrapper around ObjectToJSON(Value,[woDontStoreDefault, woFullExpand]) also able to serialize plain Exception as a simple "{"Exception":"Message"}", and append .map/.mab source code line number for ESynException

**function** ObjectToJSONFile(Value: TObject; const JSONFile: TFileName; Options: TTextWriterWriteObjectOptions=[woHumanReadable]): boolean;

*Persist a class instance into a JSON file*
- returns TRUE on success, false on error (e.g. the file name is invalid or the file is existing and could not be overwritten)
function ObjectToVariantDebug(Value: TObject; const ContextFormat: RawUTF8; const ContextArgs: array of const; const ContextName: RawUTF8='context'): variant; overload;

Will serialize any TObject into a TDocVariant debugging document
- just a wrapper around _JsonFast(ObjectToJSONDebug()) with an optional "Context":"..." text message
- if the supplied context format matches '{...}' then it will be added as a corresponding TDocVariant JSON object

function ObjectToVariantDebug(Value: TObject): variant; overload;

Will serialize any TObject into a TDocVariant debugging document
- just a wrapper around _JsonFast(ObjectToJSONDebug())

procedure ReadObject(Value: TObject; const FromContent: RawUTF8; const SubCompName: RawUTF8=''); overload;

Read an object properties, as saved by TINIWriter.WriteObject() method
- i.e. only Integer, Int64, enumerates (including boolean), floating point values and (Ansi/Wide/Unicode)String properties (excluding shortstring)
- read only the published properties of the current class level (do NOT read the properties content published in the parent classes)
- for integers, if no value is stored in FromContent, the default value is set
- this version gets the appropriate section from [Value.ClassName]
- this version doesn't handle embedded objects

procedure ReadObject(Value: TObject; From: PUTF8Char; const SubCompName: RawUTF8=''); overload;

Read an object properties, as saved by TINIWriter.WriteObject() method
- i.e. only Integer, Int64, enumerates (including boolean), floating point, variant and (Ansi/Wide/Unicode)String properties (excluding shortstring)
- read only the published properties of the current class level (do NOT read the properties content published in the parent classes)
- "From" must point to the [section] containing the object properties
- for integers and enumerates, if no value is stored in From (or From is ''), the default value from the property definition is set

function RecordReference(aTableIndex: cardinal; aID: TID): TRecordReference; overload;

Create a TRecordReference with the corresponding parameters

function RecordReference(Model: TSQLModel; aTable: TSQLRecordClass; aID: TID): TRecordReference; overload;

Create a TRecordReference with the corresponding parameters

procedure RecordRefToID(var aArray: TInt64DynArray);

Convert a dynamic array of TRecordReference into its corresponding IDs
**function SelectInClause(const PropName: RawUTF8; const Values: array of Int64; const Suffix: RawUTF8=''; ValuesInlinedMax: integer=0): RawUTF8; overload;**

*Compute 'PropName in (...)’ where clause for a SQL statement*
- if Values has no value, returns ''
- if Values has a single value, returns 'PropName=Values0' or inlined 'PropName=:(Values0):' if ValuesInlined is bigger than 1
- if Values has more than one value, returns 'PropName in (Values0,Values1,...)' or 'PropName in (:(Values0):,:(Values1):,...)' if length(Values)<ValuesInlinedMax
- PropName can be used as a prefix to the 'in ()' clause, in conjunction with optional Suffix value

**function SelectInClause(const PropName: RawUTF8; const Values: array of RawUTF8; const Suffix: RawUTF8=''; ValuesInlinedMax: integer=0): RawUTF8; overload;**

*Compute 'PropName in (...)’ where clause for a SQL statement*
- if Values has no value, returns ''
- if Values has a single value, returns 'PropName="Values0"' or inlined 'PropName=:("Values0"):' if ValuesInlined is true
- if Values has more than one value, returns 'PropName in ("Values0","Values1",...)' or 'PropName in (:("Values0"):,:("Values1"):,...)' if length(Values)<ValuesInlinedMax
- PropName can be used as a prefix to the 'in ()' clause, in conjunction with optional Suffix value

**procedure SetDefaultValuesObject(Value: TObject);**

*Set any default integer or enumerates (including boolean) published properties values for a TPersistent/TSynPersistent*
- set only the values set as "property ... default ..." at class type level
- will also reset the published properties of the nested classes

**procedure SetID(const U: RawByteString; var result: TID); overload;**

*Set the TID (=64-bit integer) value from the numerical text stored in U*
- just a redirection to SynCommons.SetInt64()

**procedure SetID(P: PUTF8Char; var result: TID); overload;**

*Set the TID (=64-bit integer) value from the numerical text stored in P^*
- just a redirection to SynCommons.SetInt64()

**procedure SetWeak(aInterfaceField: PIInterface; const aValue: IInterface);**

*Assign a Weak interface reference, to be used for circular references*
- by default setting aInterface.Field := aValue will increment the internal reference count of the implementation object: when underlying objects reference each other via interfaces (e.g. as parent and children), what causes the reference count to never reach zero, therefore resulting in memory links
- to avoid this issue, use this procedure instead
procedure SetWeakZero(aObject: TObject; aObjectInterfaceField: PIInterface; const aValue: IInterface);

{IFDEF HASINLINE}inline{IFDEF}
raise compilation Internal Error C2170 assign a Weak interface reference, which will be ZEROed (set to nil) when the corresponding object will be released
- this function is bit slower than SetWeak, but will avoid any GPF, by maintaining a list of per-instance weak interface field reference, and hook the Freelnstance virtual method in order to reset any reference to nil: Freelnstance will be overridden for this given class VMT only (to avoid unnecessary slowdown of other classes), calling the previous method afterward (so will work even with custom Freelnstance implementations)
- for faster possible retrieval, it will assign the unused vmtAutoTable VMT entry trick (just like TSQLRecord.RecordProps) - note that it will be compatible also with interfaces implemented via TSQLRecord children
- thread-safe implementation, using a per-class fast lock

function SQLFromWhere(const Where: RawUTF8): RawUTF8;

Compute the SQL corresponding to a WHERE clause
- returns directly the Where value if it starts with one the ORDER/GROUP/LIMIT/OFFSET/JOIN keywords
- otherwise, append ' WHERE ' + Where

function SQLGetOrder(const SQL: RawUTF8): RawUTF8;

Get the order table name from a SQL statement
- return the word following any 'ORDER BY' statement
- return 'RowID' if none found

function SQLWhereIsEndClause(const Where: RawUTF8): boolean;

Find out if the supplied WHERE clause starts with one of the ORDER/GROUP/LIMIT/OFFSET/JOIN keywords

function StatusCodeIsSuccess(Code: integer): boolean;

Returns true for successful HTTP status codes, i.e. in 200..399 range
- will map mainly SUCCESS (200), CREATED (201), NOCONTENT (204), PARTIALCONTENT (206), NOTMODIFIED (304) or TEMPORARYREDIRECT (307) codes
- any HTTP status not part of this range will be identified as erroneous request in the internal server statistics

function StatusCodeToErrorMsg(Code: integer): shortstring;

Convert any HTTP_* constant to an integer error code and its English text
- see @http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
- will call StatusCodeToErrorMessage()

function ToMethod(const method: RawUTF8): TSQLURIMethod;

Convert a string HTTP verb into its TSQLURIMethod enumerate

function ToText(const aGUID: TGUID): TGUIDShortString; overload;

Returns the interface name of a registered GUID, or its hexadecimal value

function TSQLRawBlobToBlob(const RawBlob: TSQLRawBlob): RawUTF8; overload;

Creates a TEXT-encoded version of blob data from a TSQLRawBlob
- TEXT will be encoded as SQLite3 BLOB literals (X'53514C697465' e.g.)
function TSQLRawBlobToBlob(RawBlob: pointer; RawBlobLength: integer): RawUTF8;
overload;

    Creates a TEXT-encoded version of blob data from a memory data
    - same as TSQLRawBlob, but with direct memory access via a pointer/byte size pair
    - TEXT will be encoded as SQLite3 BLOB literals (X'53514C697465' e.g.)

function TSQLRecordDynArrayCompare(const Item1, Item2): integer;
TDynArraySortCompare compatible function, sorting by TSQLRecord.ID

function TSQLRecordDynArrayHashOne(const Elem; Hasher: THasher): cardinal;
TDynArrayHashOne compatible function, hashing TSQLRecord.ID

function UnJSONFirstField(var P: PUTF8Char): RawUTF8;
    Get the FIRST field value of the FIRST row, from a JSON content
    - e.g. useful to get an ID without converting a JSON content into a TSQLTableJSON
    Used for DI-2.1.2 (page 2545).

function URIRequest(url, method, SendData: PUTF8Char; Resp, Head: PPUTF8Char): Int64Rec; cdecl;
    This function can be exported from a DLL to remotely access to a TSQLRestServer
    - use TSQLRestServer.ExportServer to assign a server to this function
    - return 501 HTTP_NOTIMPLEMENTED if no TSQLRestServer.ExportServer has been assigned
    - memory for Resp and Head are allocated with GlobalAlloc(): client must release this pointers
      with GlobalFree() after having retrieved their content
    - simply use TSQLRestClientURIDll to access to an exported URIRequest() function
    Used for DI-2.1.1.2.1 (page 2544).

function UrlDecodeObject(U: PUTF8Char; Upper: PAnsiChar; var ObjectInstance; Next: PPUTF8Char=nil; Options: TJSONToObjectOptions=[]): boolean;
    Decode a specified parameter compatible with URI encoding into its original object contents
    - ObjectInstance must be an existing TObject instance
    - will call interanaly JSONToObject() function to unserialize its content
    - UrlDecodeExtended('price=20.45&where=LastName%3D%27M%C3%B4net%27','PRICE=',P,@Next)
      will return Next^='where=...' and P=20.45
    - if Upper is not found, Value is not modified, and result is FALSE
    - if Upper is found, Value is modified with the supplied content, and result is TRUE

function UTF8CompareBoolean(P1, P2: PUTF8Char): PtrInt;
    Special comparison function for sorting sftBoolean UTF-8 encoded values in the SQLite3 database or JSON content

function UTF8CompareCurr64(P1, P2: PUTF8Char): PtrInt;
    Special comparison function for sorting sftCurrency UTF-8 encoded values in the SQLite3 database or JSON content

function UTF8CompareDouble(P1, P2: PUTF8Char): PtrInt;
    Special comparison function for sorting sftFloat UTF-8 encoded values in the SQLite3 database or JSON content
function UTF8CompareInt64(P1,P2: PUTF8Char): PtrInt;
  
  Special comparison function for sorting sftInteger, sftTID, sftRecordVersion
  sftTimeLog/sftModTime/sftCreateTime or sftUnixTime/sftUnixMSTime UTF-8 encoded values in
  the SQLite3 database or JSON content

function UTF8CompareISO8601(P1,P2: PUTF8Char): PtrInt;
  
  Special comparison function for sorting sftDateTime or sftDateTimeMS UTF-8 encoded values in
  the SQLite3 database or JSON content

function UTF8CompareRecord(P1,P2: PUTF8Char): PtrInt;
  
  Special comparison function for sorting ftRecord (TRecordReference/RecordRef) UTF-
  8 encoded values in the SQLite3 database or JSON content

function UTF8CompareUInt32(P1,P2: PUTF8Char): PtrInt;
  
  Special comparison function for sorting sftEnumerate, sftSet or sftID UTF-8 encoded values in
  the SQLite3 database or JSON content

function UTF8ContentNumberType(P: PUTF8Char): TSQLFieldType;
  
  Guess the number type of an UTF-8 encoded field value, as used in TSQLTable.Get()
  - if P if nil or 'null', return sftUnknown
  - will return sftInteger or sftFloat if the supplied text is a number
  - will return sftUTF8Text for any non numerical content

function UTF8ContentType(P: PUTF8Char): TSQLFieldType;
  
  Guess the content type of an UTF-8 encoded field value, as used in TSQLTable.Get()
  - if P if nil or 'null', return sftUnknown
  - otherwise, guess its type from its value characters
  - sftBlob is returned if the field is encoded as SQLite3 BLOB literals (X'53514C697465' e.g.) or with
    \uff00' magic code
  - since P is PUTF8Char, string type is sftUTF8Text only
  - sftFloat is returned for any floating point value, even if it was declared as sftCurrency type
  - sftInteger is returned for any INTEGER stored value, even if it was declared as sftEnumerate,
    sftSet, sftID, sftTID, sftRecord, sftRecordVersion, sftSessionUserID, sftBoolean,
    sftModTime/sftCreateTime/sftTimeLog or sftUnixTime/sftUnixMSTime type

procedure ValueVarToVariant(Value: PUTF8Char; ValueLen: integer; fieldType:
  TSQLFieldType; var result: TVarData; createValueTempCopy: boolean; typeInfo: pointer;
  options: TDocVariantOptions=JSON_OPTIONS_FAST);
  
  Low-level function used to convert a JSON Value into a variant, according to the property type
  - for sftObject, sftVariant, sftBlobDynArray and sftUTF8Custom, the JSON buffer may be an array
    or an object, so createValueTempCopy can create a temporary copy before parsing it in-place, to
    preserve the buffer
  - sftUnknown and sftMany will set a varEmpty (Unassigned) value
  - typeInfo may be used for sftBlobDynArray conversion to a TDocVariant array
procedure WriteObject(Value: TObject; var IniContent: RawUTF8; const Section: RawUTF8; const SubCompName: RawUTF8=''); overload;

Write an object properties, as saved by TINIWriter.WriteObject() method
  - i.e. only Integer, Int64, enumerates (including boolean), floating point values and
    (Ansi/Wide/Unicode)String properties (excluding shortstring)
  - write only the published properties of the current class level (do NOT write the properties
    content published in the parent classes)
  - direct update of INI-like content
  - for integers, value is always written, even if matches the default value

function WriteObject(Value: TObject): RawUTF8; overload;

Write an object properties, as saved by TINIWriter.WriteObject() method
  - i.e. only Integer, Int64, enumerates (including boolean), floating point values and
    (Ansi/Wide/Unicode)String properties (excluding shortstring)
  - write only the published properties of the current class level (do NOT write the properties
    content published in the parent classes)
  - return the properties as text Name=Values pairs, with no section
  - for integers, if the value matches the default value, it is not added to the result

procedure _ObjAddProps(Value: TObject; var Obj: variant); overload;

Add the property values of a TObject to a document-based object content
  - if Obj is a TDocVariant object, then all Values's published properties will be added at the root
    level of Obj

Variables implemented in the mORMot unit

AlgoDeflate: TAlgoCompress;
  Access to Zip Deflate compression in level 6 as a TSynCompress class

AlgoDeflateFast: TAlgoCompress;
  Access to Zip Deflate compression in level 1 as a TAlgoCompress class

AuthAdminDefaultPassword: RawUTF8 = DEFAULT_HASH_SYNOPSE;
  Default hashed password set by TSQLAuthGroup.InitializeTable for 'Admin' user
  - you can override this value to follow your own application expectations

AuthAdminGroupDefaultTimeout: integer = 10;
  Default timeout period set by TSQLAuthGroup.InitializeTable for 'Admin' group
  - you can override this value to follow your own application expectations

AuthGuestGroupDefaultTimeout: integer = 60;
  Default timeout period set by TSQLAuthGroup.InitializeTable for 'Guest' group
  - you can override this value to follow your own application expectations
  - note that clients will maintain the session alive using CacheFlush/_ping_

AuthSupervisorDefaultPassword: RawUTF8 = DEFAULT_HASH_SYNOPSE;
  Default hashed password set by TSQLAuthGroup.InitializeTable for 'Supervisor' user
  - you can override this value to follow your own application expectations

AuthSupervisorGroupDefaultTimeout: integer = 60;
  Default timeout period set by TSQLAuthGroup.InitializeTable for 'Supervisor' group
  - you can override this value to follow your own application expectations
  - note that clients will maintain the session alive using CacheFlush/_ping_
AuthUserDefaultPassword: RawUTF8 = DEFAULT_HASH_SYNOPSE;

*Default hashed password set by TSQLAuthGroup.InitializeTable for 'User' user*
- you can override this value to follow your own application expectations

AuthUserGroupDefaultTimeout: integer = 60;

*Default timeout period set by TSQLAuthGroup.InitializeTable for 'User' group*
- you can override this value to follow your own application expectations
- note that clients will maintain the session alive using CacheFlush/_ping_

DEFAULT_WRITEOPTIONS: array[boolean] of TTextWriterWriteObjectOptions = ([woDontStoreDefault, woSQLRawBlobAsBase64], [woDontStoreDefault, woDontStoreEmptyString, woDontStore0, woSQLRawBlobAsBase64]);

*The options used by TObjArraySerializer, TInterfacedObjectFake and TServiceMethodExecute when serializing values as JSON*
- used as DEFAULT_WRITEOPTIONS[DontStoreVoidJSON]
- you can modify this global variable to customize the whole process

ObjArraySerializers: TPointerClassHash;

*A shared list of T*ObjArray registered serializers*
- you should not access this variable, but via inline methods

ServiceContext: TServiceRunningContext;

*This thread-specific variable will be set with the currently running service context (on the server side)*
- note that in case of direct server side execution of the service, this information won't be filled, so the safest (and slightly faster) access to the TSQLRestServer instance associated with a service is to inherit your implementation class from TInjectableObjectRest, and not use this threadvar
- is set by TServiceFactoryServer.ExecuteMethod() just before calling the implementation method of a service, allowing to retrieve the current execution context - Request member is set from a client/server execution: Request.Server is the safe access point to the underlying TSQLRestServer, in such context - also consider the CurrentServiceContextServer function to retrieve directly the running TSQLRestServer (if any)
- its content is reset to zero out of the scope of a method execution
- when used, a local copy or a PServiceRunningContext pointer should better be created, since accessing a threadvar has a non negligible performance cost - for instance, if you want to use a "with" statement:

```delphi
with PServiceRunningContext(@ServiceContext)^ do
  ... access TServiceRunningContext members
```

or as a local variable:

```delphi
var context: PServiceRunningContext;
inContentType: RawUTF8;
begin
  context := @ServiceContext; // threadvar access once
  ...
inContentType := context.Request.Call^.InBodyType;
end;
```

- when accessed from a package, use function CurrentServiceContext() instead, to circumvent a Delphi RTL/compiler restriction (bug?)

SERVICELOG_WRITEOPTIONS: TTextWriterWriteObjectOptions = [woDontStoreDefault, woDontStoreEmptyString, woDontStore0, woHideSynPersistentPassword];

*The options used by TServiceFactoryServer.OnLogRestExecuteMethod*
- you can modify this global variable to customize the whole process
SetThreadNameLog: TSynLogClass = TSQLLog;

TSQLLogClass used by overridden SetThreadName() function to name the thread

SETTINGS_WRITEOPTIONS: TTextWriterWriteObjectOptions = [woHumanReadable,
woStoreStoredFalse, woHumanReadableFullSetsAsStar,
woHumanReadableEnumSetAsComment, woInt64AsHex];

The options used by TSynJsonFileSettings.SaveIfNeeded
- you can modify this global variable to customize the whole process

SQLite3Log: TSynLogClass = TSQLLog;

TSQLLog class is used for logging for all our ORM related functions
- this global variable can be used to customize it for the whole process
- each TSQLRest.LogClass property is set by default to this SQLite3Log
- you can override the TSQLRest.LogClass property value to customize it for a given REST instance

StatusCodeToErrorMessage: procedure(Code: integer; var result: RawUTF8);

Convert any HTTP_* constant to a short English text
- see @http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
- if SynCrtSock is linked (i.e. set mORMotHttpClient or mORMotHttpServer), SynCrtSock
StatusCodeToReason() function will be assigned, which is somewhat faster, and more complete

USEFASTMMAALLOC: boolean = false;

If this variable is TRUE, the URIRequest() function won't use Win32 API GlobalAlloc() function, but fastest native Getmem()
- can be also useful for debug

Used for DI-2.1.1.2.1 (page 2544).
27.53. mORMotDB.pas unit

Purpose: Virtual Tables for external DB access for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the mORMotDB unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/cache/refresh/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

mORMotDB class hierarchy

Objects implemented in the mORMotDB unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLRestStorageExternal</td>
<td>REST server with direct access to a SynDB-based external database</td>
<td>2277</td>
</tr>
<tr>
<td>TSQLVirtualTableCursorExternal</td>
<td>A Virtual Table cursor for reading a TSQLDBStatement content</td>
<td>2279</td>
</tr>
<tr>
<td>TSQLVirtualTableExternal</td>
<td>A SynDB-based virtual table for accessing any external database</td>
<td>2280</td>
</tr>
</tbody>
</table>
TSQLRestStorageExternal = class(TSQLRestStorage)

REST server with direct access to a SynDB-based external database
- handle all REST commands, using the external SQL database connection, and prepared statements
- is used by TSQLRestServer.URI for faster RESTful direct access
- for JOINed SQL statements, the external database is also defined as a SQLite3 virtual table, via the TSQLVirtualTableExternal[Cursor] classes

constructor Create(aClass: TSQLRecordClass; aServer: TSQLRestServer); override;
Initialize the remote database connection
- you should not use this, but rather call VirtualTableExternalRegister()
- RecordProps.ExternalDatabase will map the associated TSQLDBConnectionProperties
- RecordProps.ExternalTableName will retrieve the real full table name, e.g. including any database schema prefix

destructor Destroy; override;
Finalize the remote database connection

function ComputeSQL(const Prepared: TSQLVirtualTablePrepared): RawUTF8;
Compute the SQL query corresponding to a prepared request
- can be used internally e.g. for debugging purposes

class function ConnectionProperties(aClass: TSQLRecordClass; aServer: TSQLRestServer): TSQLDBConnectionProperties; overload;
Retrieve the external database connection associated to a TSQLRecord
- just map aServer.StaticVirtualTable[] and will return nil if not a TSQLRestStorageExternal

function CreateSQLMultiIndex(Table: TSQLRecordClass; const FieldNames: array of RawUTF8; Unique: boolean; IndexName: RawUTF8=''): boolean; override;
Create one index for all specific FieldNames at once
- this method will in fact call the SQLAddIndex method, if the index is not already existing
- for databases which do not support indexes on BLOB fields (i.e. all engine but SQLite3), such FieldNames will be ignored

function EngineDelete(TableModelIndex: integer; ID: TID): boolean; override;
Delete a row, calling the external engine with SQL
- made public since a TSQLRestStorage instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

function EngineUpdateField(TableModelIndex: integer; const SetFieldName, SetValue, WhereFieldName, WhereValue: RawUTF8): boolean; override;
Update a field value of the external database

function EngineUpdateFieldIncrement(TableModelIndex: integer; ID: TID; const FieldName: RawUTF8; Increment: Int64): boolean; override;
Update a field value of the external database

class function Instance(aClass: TSQLRecordClass; aServer: TSQLRestServer): TSQLRestStorageExternal;
Retrieve the REST server instance corresponding to an external TSQLRecord
- just map aServer.StaticVirtualTable[] and will return nil if not a TSQLRestStorageExternal
- you can use it e.g. to call MapField() method in a fluent interface
function RetrieveBlobFields(Value: TSQLRecord): boolean; override;
  Overridden method for direct external database engine call

function SearchField(const FieldName: RawUTF8; FieldValue: Int64; out ResultID: TIDDynArray): boolean; overload; override;
  Search for a numerical field value
  - return true on success (i.e. if some values have been added to ResultID)
  - store the results into the ResultID dynamic array

function SearchField(const FieldName, FieldValue: RawUTF8; out ResultID: TIDDynArray): boolean; overload; override;
  Search for a field value, according to its SQL content representation
  - return true on success (i.e. if some values have been added to ResultID)
  - store the results into the ResultID dynamic array

function TableHasRows(Table: TSQLRecordClass): boolean; override;
  Overridden method for direct external database engine call

function TableRowCount(Table: TSQLRecordClass): Int64; override;
  Overridden method for direct external database engine call

function TransactionBegin(aTable: TSQLRecordClass; SessionID: cardinal=1): boolean; override;
  Begin a transaction (implements REST BEGIN Member)
  - to be used to speed up some SQL statements like Insert/Update/Delete
  - must be ended with Commit on success
  - must be aborted with Rollback if any SQL statement failed
  - return true if no transaction is active, false otherwise

function UpdateBlobFields(Value: TSQLRecord): boolean; override;
  Overridden method for direct external database engine call

procedure Commit(SessionID: cardinal=1; RaiseException: boolean=false); override;
  End a transaction (implements REST END Member)
  - write all pending SQL statements to the external database

procedure EndCurrentThread(Sender: TThread); override;
  This method is called by TSQLRestServer.EndCurrentThread method just before a thread is finished to ensure that the associated external DB connection will be released for this thread
  - this overridden implementation will clean thread-specific connections, i.e. call TSQLDBConnectionProperties.ThreadSafe.EndCurrentThread method
  - this method shall be called directly, nor from the main thread

procedure EngineAddForceSelectMaxID;
  Reset the internal cache of external table maximum ID
  - next EngineAdd/BatchAdd will execute SELECT max(ID) FROM externaltable
  - is a lighter alternative to EngineAddUseSelectMaxID=TRUE, since this method may be used only once, when some records have been inserted into the external database outside this class scope (e.g. by legacy code)

procedure RollBack(SessionID: cardinal=1); override;
  Abort a transaction (implements REST ABORT Member)
  - restore the previous state of the database, before the call to TransactionBegin
property EngineAddForcedID: TID read fEngineAddForcedID write fEngineAddForcedID;
Disables internal ID generation for INSERT
- by default, a new ID will be set (either with 'select max(ID)' or via the OnEngineLockedNextID event)
- if the client supplies a forced ID within its JSON content, it would be used for adding
- define this property to a non 0 value if no such ID is expected to be supplied, but a fixed "fake ID" is returned by the Add() method; at external DB level, no such ID field would be computed nor set at INSERT - this feature may be useful when working with a legacy database - of course any ID-based ORM method would probably fail to work

property EngineAddUseSelectMaxID: Boolean read fEngineAddUseSelectMaxID write fEngineAddUseSelectMaxID;
By default, any INSERT will compute the new ID from an internal variable
- it is very fast and reliable, unless external IDs can be created outside this engine
- you can set EngineAddUseSelectMaxID=true to execute a slower 'select max(ID)' from TableName' SQL statement before each EngineAdd()
- a lighter alternative may be to call EngineAddForceSelectMaxID only when required, i.e. when the external DB has just been modified by a third-party/legacy SQL process

property OnEngineAddComputeID: TOnEngineAddComputeID read fOnEngineAddComputeID write fOnEngineAddComputeID;
Define an alternate method of compute the ID for INSERT
- by default, a new ID will be with 'select max(ID)', and an internal counter (unless EngineAddUseSelectMaxID is true)
- you can specify a custom callback, which may compute the ID as expected (e.g. using a SQL sequence)

property Properties: TSQLDBConnectionProperties read GetConnectionProperties;
The associated external SynDB database connection properties

TSQLVirtualTableCursorExternal = class(TSQLVirtualTableCursor)
A Virtual Table cursor for reading a TSQLDBStatement content
- this is the cursor class associated to TSQLVirtualTableExternal

destructor Destroy; override;
Finalize the external cursor by calling ReleaseRows

function Column(aColumn: integer; var aResult: TSQLVar): boolean; override;
Called to retrieve a column value of the current data row
- if aColumn=VIRTUAL_TABLE_ROWID_COLUMN(-1), will return the row ID as varInt64 into aResult
- will return false in case of an error, true on success

function HasData: boolean; override;
Called after Search() to check if there is data to be retrieved
- should return false if reached the end of matching data

function Next: boolean; override;
Called to go to the next row of matching data
- should return false on low-level database error (but true in case of a valid call, even if HasData will return false, i.e. no data match)
function Search(const Prepared: TSQLVirtualTablePrepared): boolean; override;

Called to begin a search in the virtual table, creating a SQL query
- the TSQLVirtualTablePrepared parameters were set by TSQLVirtualTable.Prepare and will contain both WHERE and ORDER BY statements (retrieved by x_BestIndex from a TSQLite3IndexInfo structure)
- Prepared will contain all prepared constraints and the corresponding expressions in the Where[].Value field
- will move cursor to first row of matching data
- will return false on low-level database error (but true in case of a valid call, even if HasData will return false, i.e. no data match)
- all WHERE and ORDER BY clauses are able to be translated into a plain SQL statement calling the external DB engine
- will create the internal fStatement from a SQL query, bind the parameters, then execute it, ready to be accessed via HasData/Next

property SQL: RawUTF8 read fSQL;

Read-only access to the SELECT statement

TSQLVirtualTableExternal = class(TSQLVirtualTable)

A SynDB-based virtual table for accessing any external database
- for ORM access, you should use VirtualTableExternalRegister method to associate this virtual table module to any TSQLRecord class
- transactions are handled by this module, according to the external database

function Delete(aRowID: Int64): boolean; override;

Called to delete a virtual table row
- returns true on success, false otherwise

function Drop: boolean; override;

Called when a DROP TABLE statement is executed against the virtual table
- returns true on success, false otherwise

function Insert(aRowID: Int64; var Values: TSQLVarDynArray; out insertedRowID: Int64): boolean; override;

Called to insert a virtual table row content
- column order follows the Structure method, i.e. StoredClassProps.Fields[] order
- returns true on success, false otherwise
- returns the just created row ID in insertedRowID on success

function Prepare(var Prepared: TSQLVirtualTablePrepared): boolean; override;

Called to determine the best way to access the virtual table
- will prepare the request for TSQLVirtualTableCursor.Search()
- this overridden method will let the external DB engine perform the search, using a standard SQL "SELECT * FROM .. WHERE .. ORDER BY .." statement
- in Where[], Expr must be set to not 0 if needed for Search method, and OmitCheck always set to true since double check is not necessary
- OmitOrderBy will be set to true since double sort is not necessary
- EstimatedCost/EstimatedRows will receive the estimated cost, with lowest value if fStatic.fFieldsExternal[].ColumnIndexed is set (i.e. if column has an index)
function Update(oldRowID, newRowID: Int64; var Values: TSQLVarDynArray): boolean;
override;

Called to update a virtual table row content
- column order follows the Structure method, i.e. StoredClassProps.Fields[] order
- returns true on success, false otherwise

class procedure GetTableModuleProperties(var aProperties: TVirtualTableModuleProperties); override;

Returns the main specifications of the associated TSQLVirtualTableModule
- this is a read/write table, without transaction (yet), associated to the
  TSQLVirtualTableCursorExternal cursor type, with 'External' as module name and
  TSQLRestStorageExternal as the related static class
- no particular class is supplied here, since it will depend on the associated Static
  TSQLRestStorageExternal instance

Types implemented in the mORMotDB unit

TOnEngineAddComputeID = function(Sender: TSQLRestStorageExternal; var Handled: Boolean): TID of object;

Event handler called to customize the computation of a new ID
- should set Handled=TRUE if a new ID has been computed and returned
- Handled=False would let the default ID computation take place
- note that execution of this method would be protected by a mutex, so it would be thread-safe

TVirtualTableExternalRegisterOptions = set of TVirtualTableExternalRegisterOption;

All possible options for VirtualTableExternalRegisterAll/TSQLRestExternalDBCreate
- by default, TSQLAuthUser and TSQLAuthGroup tables will be handled via the external DB, but you
can avoid it for speed when handling session and security by setting
regDoNotRegisterUserGroupTables
- you can set regMapAutoKeywordFields to ensure that the mapped field names won't conflict with
a SQL reserved keyword on the external database by mapping a name with a trailing '_' character for
the external column
- regClearPoolOnConnectionIssue will call ClearConnectionPool when a connection-linked exception
is discovered

Functions or procedures implemented in the mORMotDB unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLRestExternalDBCreate</td>
<td>Create a new TSQLRest instance, and possibly an external database, from its Model and stored values</td>
<td>2282</td>
</tr>
<tr>
<td>VirtualTableExternalMap</td>
<td>Register one table of the model to be external, with optional mapping</td>
<td>2282</td>
</tr>
<tr>
<td>VirtualTableExternalRegister</td>
<td>Register several tables of the model to be external</td>
<td>2283</td>
</tr>
</tbody>
</table>
### Functions or procedures

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VirtualTableExternalRegister</td>
<td>Register on the Server-side an external database for an ORM class</td>
<td>2283</td>
</tr>
<tr>
<td>VirtualTableExternalRegisterAll</td>
<td>Register all tables of the model to be external, with some options</td>
<td>2284</td>
</tr>
<tr>
<td>VirtualTableExternalRegisterAll</td>
<td>Register all tables of the model to be external</td>
<td>2284</td>
</tr>
</tbody>
</table>

```pascal
function TSQLRestExternalDBCreate(aModel: TSQLModel; aDefinition: TSynConnectionDefinition; aHandleAuthentication: boolean; aExternalOptions: TVirtualTableExternalRegisterOptions): TSQLRest; overload;

Create a new TSQLRest instance, and possibly an external database, from its Model and stored values
- if aDefinition.Kind matches a TSQLRest registered class, one new instance of this kind will be created and returned
- if aDefinition.Kind is a registered TSQLDBConnectionProperties class name, it will instantiate an in-memory TSQLRestServerDB or a TSQLRestServerFullMemory instance, then call VirtualTableExternalRegisterAll() on this connection
- will return nil if the supplied aDefinition does not match any registered TSQLRest or TSQLDBConnectionProperties types
```

```pascal
function VirtualTableExternalMap(aModel: TSQLModel; aClass: TSQLRecordClass; aExternalDB: TSQLDBConnectionProperties; const aExternalTableName: RawUTF8=''; aMapping: TSQLRecordPropertiesMappingOptions=[]): PSQLRecordPropertiesMapping;

Register one table of the model to be external, with optional mapping
- this method would allow to chain MapField() or MapAutoKeywordFields definitions, in a fluent interface:
```
**function** VirtualTableExternalRegister(aModel: TSQLModel; aClass: TSQLRecordClass; aExternalDB: TSQLDBConnectionProperties; const aExternalTableName: RawUTF8=''; aMappingOptions: TSQLRecordPropertiesMappingOptions=[]): boolean; overload;

Register on the Server-side an external database for an ORM class
- will associate the supplied class with a TSQLVirtualTableExternal module (calling
aModel.VirtualTableRegister method), even if the class does not inherit from
TSQLRecordVirtualTableAutoID (it can be any plain TSQLRecord or TSQLRecordMany sub-class for
instance)
- note that TSQLModel.Create() will reset all supplied classes to be defined as non virtual (i.e.
Kind=rSQLite3)
- this function shall be called BEFORE TSQLRestServer.Create (the server-side ORM must know if
the database is to be managed as internal or external)
- this function (and the whole unit) is NOT to be used on the client-side
- the TSQLDBConnectionProperties instance should be shared by all classes, and released globally
when the ORM is no longer needed
- the full table name, as expected by the external database, could be provided here
(SQLTableName will be used internally as table name when called via the associated SQLite3
Virtual Table) - if no table name is specified (''), will use SQLTableName (e.g. 'Customer' for
'TSQLCustomer')
- typical usage is therefore for instance:
  Props := TOleDBMSSQLConnectionProperties.Create('.\SQLEXPRESS','AdventureWorks2008R2','','');
  Model := TSQLModel.Create(TSQLCustomer,'root');
  VirtualTableExternalRegister(Model,TSQLCustomer,Props,'Sales.Customer');
  Server := TSQLRestServerDB.Create(aModel,'application.db',true)
- the supplied aExternalDB parameter is stored within aClass.RecordProps, so the instance must
  stay alive until all database access to this external table is finished (e.g. use a private/protected
  property)
- aMappingOptions can be specified now, or customized later
- server-side may omit a call to VirtualTableExternalRegister() if the need of an internal database is
  expected: it will allow custom database configuration at runtime, depending on the customer's
  expectations (or license)
- after registration, you can tune the field-name mapping by calling
  aModel.Props[aClass].ExternalDB.MapField(..)

**function** VirtualTableExternalRegister(aModel: TSQLModel; const aClass: array of
TSQLRecordClass; aExternalDB: TSQLDBConnectionProperties; aMappingOptions:
TSQLRecordPropertiesMappingOptions=[]): boolean; overload;

Register several tables of the model to be external
- just a wrapper over the overloaded VirtualTableExternalRegister() method

**function** VirtualTableExternalRegisterAll(aModel: TSQLModel; aExternalDB:
TSQLDBConnectionProperties; DoNotRegisterUserGroupTables: boolean=false;
ClearPoolOnConnectionIssue: boolean=false): boolean; overload;

Register all tables of the model to be external
- mainly for retro-compatibility with existing code
- just a wrapper around the VirtualTableExternalRegisterAll() overloaded function with some
boolean flags instead of TVirtualTableExternalRegisterOptions
function VirtualTableExternalRegisterAll(aModel: TSQLModel; aExternalDB: TSQLDBConnectionProperties; aExternalOptions: TVirtualTableExternalRegisterOptions): boolean; overload;

Register all tables of the model to be external, with some options
- by default, all tables are handled by the SQLite3 engine, unless they are explicitly declared as external via VirtualTableExternalRegister: this function can be used to register all tables to be handled by an external DB
- this function shall be called BEFORE TSQLRestServer.Create (the server-side ORM must know if the database is to be managed as internal or external)
- this function (and the whole unit) is NOT to be used on the client-side
- the TSQLDBConnectionProperties instance should be shared by all classes, and released globally when the ORM is no longer needed
- by default, TSQLAuthUser and TSQLAuthGroup tables will be handled via the external DB, but you can avoid it for speed when handling session and security by setting regDoNotRegisterUserGroupTables in aExternalOptions
- other aExternalOptions can be defined to tune the ORM process e.g. about mapping or connection loss detection
- after registration, you can tune the field-name mapping by calling aModel.Props[aClass].ExternalDB.MapField(..)
27.54. mORMotDDD.pas unit

*Purpose:* Domain-Driven-Design toolbox for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the mORMotDDD unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrtSock</td>
<td>Classes implementing TCP/UDP/HTTP client and server protocol</td>
<td>1083</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
<tr>
<td></td>
<td>- implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>
Objects implemented in the mORMotDDD unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECQRIException</td>
<td>Exception type linked to CQRS repository service methods</td>
<td>2287</td>
</tr>
<tr>
<td>EDDDEException</td>
<td>Abstract ancestor for all Domain-Driven Design related Exceptions</td>
<td>2287</td>
</tr>
<tr>
<td>EDDDIInfraException</td>
<td>Abstract ancestor for any Domain-Driven Design infrastructure Exceptions</td>
<td>2287</td>
</tr>
<tr>
<td>EDDDRepository</td>
<td>Abstract ancestor for all persistence/repository related Exceptions</td>
<td>2287</td>
</tr>
<tr>
<td>IAdministratedDaemon</td>
<td>Generic interface, to manage a service/daemon instance from an executable</td>
<td>2288</td>
</tr>
<tr>
<td>IAdministratedDaemonAsProxy</td>
<td>Any service/daemon implementing this interface would be able to redirect all the administration process to another service/daemon</td>
<td>2289</td>
</tr>
<tr>
<td>ICQRSService</td>
<td>Generic interface, to be used for CQRS I<em>Query and I</em>Command types definition</td>
<td>2287</td>
</tr>
<tr>
<td>IDDDSettingsStorable</td>
<td>Allow persistence of any TObject settings storage</td>
<td>2289</td>
</tr>
<tr>
<td>IMonitored</td>
<td>Generic interface, to be used so that you may retrieve a running state</td>
<td>2288</td>
</tr>
<tr>
<td>IMonitoredDaemon</td>
<td>Generic interface, to be used so that you may manage a service/daemon instance</td>
<td>2288</td>
</tr>
<tr>
<td>TCQRSQueryObjectRest</td>
<td>Abstract CQRS class tied to a TSQLRest instance for low-level persistence</td>
<td>2295</td>
</tr>
<tr>
<td>TCQRSService</td>
<td>To be inherited to implement CQRS I<em>Query or I</em>Command services extended error process</td>
<td>2290</td>
</tr>
<tr>
<td>TCQRSServiceAsynchAck</td>
<td>Used to acknowledge asynchronous CQRS Service calls</td>
<td>2290</td>
</tr>
<tr>
<td>TCQRSServiceSubscribe</td>
<td>A CQRS Service, which maintains an internal list of &quot;Subscribers&quot;</td>
<td>2290</td>
</tr>
</tbody>
</table>
TCQRSServiceSynch
A CQRS Service, ready to implement a set of synchronous (blocking) commands over an asynchronous (non-blocking) service

TODDAE.AdministratedDaemon
Abstract class to implement an administrable service/daemon

TODDAE.AdministratedDaemonMonitor
Abstract class to monitor an administrable service/daemon

TODDAE.AdministratedDaemonRest
Abstract class to implement a TSQLRest-based administrable service/daemon

TODDAE.AdministratedThreadDaemon
Abstract class to implement an TThread-based administrable service/daemon

TODDApplication
Abstract class for implementing an Application Layer service

TODDMonitoredDaemon
Abstract class using several process threads and with monitoring abilities

TODDMonitoredDaemonProcess
Abstract process thread class with monitoring abilities

TODDMonitoredDaemonProcessRest
Abstract process thread class with monitoring abilities, using the ORM for pending tasks persistence

TODDRepositoryRestCommand
Abstract class to implement I*Command interface using ORM's TSQLRecord

TODDRepositoryRestFactory
Implement a DDD Entity factory over one ORM's TSQLRecord

TODDRepositoryRestManager
Home repository of several DDD Entity factories using REST storage

TODDRepositoryRestQuery
Abstract repository class to implement I*Query interface using RESTful ORM

EDDDException = class(ESynException)
Abstract ancestor for all Domain-Driven Design related Exceptions

ECQRSException = class(EDDDException)
Exception type linked to CQRS repository service methods

EDDDEntraException = class(EDDDException)
Abstract ancestor for any Domain-Driven Design infrastructure Exceptions

ICQRSService = interface(IInvokable)
Generic interface, to be used for CQRS I*Query and I*Command types definition
- TCQRSService class will allow to easily implement LastError* members
- all CQRS services, which may be executed remotely, would favor a function result as TCQRSResult enumeration for error handling, rather than a local Exception, which is not likely to be transferred easily on consumer side
function GetLastError: TCQRSResult;
    *Should return the last error as an enumerate*
    - when stubed or mocked via TInterfaceStub, any method interface would return 0, i.e. cqrsSuccess by default, to let the test pass

function GetLastErrorInfo: variant;
    *Should return addition information for the last error*
    - may be a plain string, or a JSON document stored as TDocVariant

IMonitored = interface(IInvokable)
Generic interface, to be used so that you may retrieve a running state

function RetrieveState(out Status: variant): TCQRSResult;
    *Retrieve the current status of the instance*
    - the status is returned as a schema-less value (typically a TDocVariant document), which may contain statistics about the current processing numbers, timing and throughput

IMonitoredDaemon = interface(IMonitored)
Generic interface, to be used so that you may manage a service/daemon instance

function Start: TCQRSResult;
    *Launch the service/daemon*
    - should first stop any previous running instance (so may be used to restart a service on demand)

function Stop(out Information: variant): TCQRSResult;
    *Abort the service/daemon, returning statistics about the whole execution*

IAdministratedDaemon = interface(IMonitoredDaemon)
Generic interface, to manage a service/daemon instance from an executable
- in addition to Start/Stop methods, Halt would force the whole executable to abort its execution,
- SubscribeLog allows log monitoring, and DatabaseList/DatabaseExecute remote SQL/SOA execution on one or several logical REST servers
- those methods would allow a single administration daemon (installed e.g. as a Windows Service) to be able to launch and monitor child processes as individual executables, or via a custom DDD's ToolsAdmin tool
- since SubscribeLog() uses a callback, this REST server should be published via supported transmission protocol, e.g. WebSockets

function DatabaseExecute(const DatabaseName,SQL: RawUTF8): TServiceCustomAnswer;
    *Execute a SQL query on an internal database*
    - the database name should match one existing in the DatabaseList
    - the supplied SQL parameter may be #cmd internal commands: in this case, the database name may not be mandatory
    - will return JSON most of the time, but may return binary if needed
function DatabaseList: TRawUTF8DynArray;

Returns a list of internal database names, exposed by this daemon
- in practice, each database name should identify a TSQLRest instance
- the database name should be supplied to DatabaseExecute() as target

function DatabaseTables(const DatabaseName: RawUTF8): TRawUTF8DynArray;

Returns a list of tables, stored in the internal database names
- the database name should match one existing in the DatabaseList
- in practice, returns all TSQLRecord table of the TSQLRest instance

function Halt(out Information: variant): TCQRSResult;

Will Stop the service/daemon process, then quit the executable
- the returned Information and TCQRSResult are passed directly from the Stop() method

procedure CallbackReleased(const callback: IInvokable; const interfaceName: RawUTF8);

Will be called when a callback is released on the client side
- this method matches IServiceWithCallbackReleased signature
- will be used to unsubscribe any previous ISynLogCallback notification

procedure SubscribeLog(const Level: TSynLogInfos; const Callback: ISynLogCallback; ReceiveExistingKB: cardinal);

Used to subscribe for real-time remote log monitoring
- allows to track the specified log events, with a callback
- you can specify a number of KB of existing log content to send to the monitoring tool, before the actual real-time process: Callback.Log() would be called first with Level=sllNone and all the existing text

IAdministratedDaemonAsProxy = interface(IAdministratedDaemon)
Any service/daemon implementing this interface would be able to redirect all the administration process to another service/daemon
- i.e. would work as a safe proxy service, over several networks

function StartProxy(const aDDDRestClientSettings: variant): TCQRSResult;

Allows to connect to another service/daemon IAdministratedDaemon
- detailed connection definition would be supplied as a TDocVariantData object, serialized from dddInfraApp.pas TDDDRestClientSettings

IDDDSettingsStorable = interface(IInterface)
Allow persistence of any TObject settings storage

procedure StoreIfUpdated;

Persist the settings if needed
- will call the virtual InternalPersist method
TCQRSService = class(TInjectableObject)

To be inherited to implement CQRS I*Query or I*Command services extended error process
- you should never assign directly a cqrs* value to a method result, but rather use the
  CqrsBeginMethod/CqrsSetResultMsg/CqrsSetResult methods provided by this class:
  
  function TMyService.MyMethod: TCQRSResult;
  begin
    CqrsBeginMethod(qsNone,result); // reset the error information to cqrsUnspecifiedError
    ... // do some work
    if error then
      CqrsSetResultMsg(cqrsUnspecifiedError, 'Oups! For %', [name], result) else
      CqrsSetResult(cqrsSuccess,result); // instead of result := cqrsSuccess
  end;

- the methods are implemented as a simple state machine, following the TCQRSQueryAction and
  TCQRSQueryState definitions
- warning: by definition, fLastError* access is NOT thread-safe so the
  CqrsBeginMethod/CqrsSetResult feature should be used in a single context

constructor Create; override;
  Initialize the instance

destructor Destroy; override;
  Finalize the instance

property Action: TCQRSQueryAction read fAction;
  The action currently processing

property LastError: TCQRSResult read GetLastError;
  The last error, as an enumerate

property LastErrorInfo: variant read GetLastErrorInfo;
  The last error extended information, as a string or TDocVariant

property Log: TSynLogFamily read fLog write fLog;
  Where logging should take place

property State: TCQRSQueryState read fState;
  Current step of the TCQRSService state machine

TCQRSServiceSubscribe = class(TCQRSService)

A CQRS Service, which maintains an internal list of "Subscribers"
- allow to notify in cascade when a callback is released

TCQRSServiceSynch = class(TInterfacedObject)

A CQRS Service, ready to implement a set of synchronous (blocking) commands over an
asynchronous (non-blocking) service
- you may use this class e.g. at API level, over a blocking REST server, and communicate with the
  Domain event-driven services via asynchronous calls
- this class won't inherit from TCQRSService, since it would be called from multiple threads at
  once, so all CQRSSetResult() methods would fail

TCQRSServiceAsynchAck = class(TInterfacedObject)

Used to acknowledge asynchronous CQRS Service calls
- e.g. to implement TCQRSServiceSynch
EDDDRepository = class(ESynException)

Abstract ancestor for all persistence/repository related Exceptions

constructor CreateUTF8(Caller: TDDDRepositoryRestFactory; const Format: RawUTF8;
const Args: array of const);

Constructor like FormatUTF8() which will also serialize the caller info

TDDDRepositoryRestManager = class(TObject)

Home repository of several DDD Entity factories using REST storage
- this shared class will be can to manage a service-wide repositories, e.g. manage actual
I*Query/I*Command implementation classes across a set of TSQLRest instances
- is designed to optimize BATCH or transactional process

destructor Destroy; override;

Finalize all factories

function AddFactory( const aInterface: TGUID; aImplementation:
TDDDRepositoryRestClass; aAggregate: TClass; aRest: TSQLRest; aTable:
TSQLRecordClass; const TableAggregatePairs: array of RawUTF8):
TDDDRepositoryRestFactory;

Register one DDD Entity repository over an ORM's TSQLRecord
- will raise an exception if the aggregate has already been defined

function GetFactory(const aInterface: TGUID): TDDDRepositoryRestFactory;

Retrieve the registered Factory definition of a given DDD Entity
- raise an EDDDRepository exception if the TPersistence class is unknown

function GetFactoryIndex(const aInterface: TGUID): integer;

Retrieve the registered definition of a given DDD Entity in Factory[]
- returns -1 if the TPersistence class is unknown

property Factory: TDDDRepositoryRestFactoryObjArray read fFactory;

Read-only access to all defined DDD Entity factories

TDDDRepositoryRestFactory = class(TInterfaceResolverForSingleInterface)

Implement a DDD Entity factory over one ORM's TSQLRecord
- it will centralize some helper classes and optimized class mapping
- the Entity class may be defined as any TPersistent or TSynPersistent, with an obvious preference
for TSynPersistent and TSynAutoCreateFields classes

constructor Create( const aInterface: TGUID; aImplementation:
TDDDRepositoryRestClass; aAggregate: TClass; aRest: TSQLRest; aTable:
TSQLRecordClass; aOwner: TDDDRepositoryRestManager=nil); reintroduce; overload;

Initialize the DDD Aggregate factory using a mORMot ORM class
- this overloaded constructor does not expect any custom fields
- any missing or unexpected field on any side will just be ignored
constructor Create( const aInterface: TGUID; aImplementation: TDDDRepositoryRestClass; aAggregate: TClass; aRest: TSQLRest; aTable: TSQLRecordClass; const TableAggregatePairs: array of RawUTF8; aOwner: TDDDRepositoryRestManager=nil); reintroduce; overload;

Initialize the DDD Aggregate factory using a mORMot ORM class
- by default, field names should match on both sides - but you can specify a custom field mapping as TSQLRecord,Aggregate pairs
- any missing or unexpected field on any side will just be ignored

destructor Destroy; override;

Finalize the factory

function AggregateCreate: TObject;
Create a new DDD Aggregate instance

function AggregateFilterAndValidate(aAggregate: TObject; aInvalidFieldIndex: PInteger=nil; aValidator: PSynValidate=nil): RawUTF8; virtual;
Perform filtering and validation on a supplied DDD Aggregate
- all logic defined by AddFilterOrValidate() will be processed

function AggregateToJSON(aAggregate: TObject; ORMappedFields: boolean; aID: TID): RawUTF8; overload;
Serialize a DDD Aggregate as JSON RawUTF8

procedure AddFilterOrValidate(const aFieldNames: array of RawUTF8; aFilterOrValidate: TSynFilterOrValidate; aFieldNameFlattened: boolean=false);
virtual;
Register a custom filter or validator to some Aggregate's fields
- once added, the TSynFilterOrValidate instance will be owned to this factory, until it is released
- the field names should be named from their full path (e.g. 'Email' or 'Address.Country.Iso') unless aFieldNameFlattened is TRUE, which will expect ORM-like naming (e.g. 'Address.Country')
- if '*' is specified as field name, it will be applied to all text fields, so the following will ensure that all text fields will be trimmed for spaces:
  AddFilterOrValidate(['*'],TSynFilterTrim.Create);
- filters and validators will be applied to a specified aggregate using AggregateFilterAndValidate() method
- the same filtering classes as with the ORM can be applied to DDD's aggregates, e.g. TSynFilterUpperCase, TSynFilterLowerCase or TSynFilterTrim
- the same validation classes as with the ORM can be applied to DDD's aggregates, e.g. TSynValidateText.Create for a void field, TSynValidateText.Create('MinLength:5') for a more complex test (including custom password strength validation if TSynValidatePassWord is not enough), TSynValidateIPAddress.Create or TSynValidateEmail.Create for some network settings, or TSynValidatePattern.Create()
- you should not define TSynValidateUniqueField here, which could't be checked at DDD level, but rather set a "stored AS_UNIQUE" attribute in the corresponding property of the TSQLRecord type definition

procedure AggregateClear(aAggregate: TObject);
Clear all properties of a given DDD Aggregate

procedure AggregateFromTable(aSource: TSQLRecord; aAggregate: TObject);
Convert a ORM TSQLRecord instance into a DDD Aggregate
procedure AggregatesFromTableFill(aSource: TSQLRecord; var aAggregateObjArray);

Convert ORM TSQLRecord.FillPrepare instances into a DDD Aggregate ObjArray

procedure AggregateToJSON(aAggregate: TObject; W: TJSONSerializer; ORMMappedFields: boolean; aID: TID); overload;

Serialize a DDD Aggregate as JSON
- you can optionally force the generated JSON to match the mapped TSQLRecord fields, so that it would be compatible with ORM's JSON

procedure AggregateToTable(aAggregate: TObject; aID: TID; aDest: TSQLRecord);

Convert a DDD Aggregate into an ORM TSQLRecord instance

class procedure ComputeSQLRecord(const aAggregate: array of TClass;
DestinationSourceCodeFile: TFileName = '...');

Will compute the ORM TSQLRecord* source code type definitions corresponding to DDD aggregate objects into a supplied file name
- will generate one TSQLRecord* per aggregate class level, following the inheritance hierarchy
- dedicated DDD types will be translated into native ORM types (e.g. RawUTF8)
- if no file name is supplied, it will generate a dddsqlrecord.inc file in the executable folder
- could be used as such:
  TDDDRepositoryRestFactory.ComputeSQLRecord([TPersonContactable, TAuthInfo]);
- once created, you may refine the ORM definition, by adding
  ... read f... write f... stored AS_UNIQUE;
for fields which should be unique, and/or
  ... read f... write f... index #;
to specify an optional textual field width (VARCHAR n) for SQL storage
- most advanced ORM-level filters/validators, or low-level implementation details (like the Sqlite3 collation) may be added by overriding this method:
protected
  class procedure InternalDefineModel(Props: TSQLRecordProperties); override;
...
class procedure TSQLRecordMyAggregate.InternalDefineModel(
Props: TSQLRecordProperties);
begin
  AddFilterNotVoidText(['HashedPassword']);
  Props.SetCustomCollation('Field', 'BINARY');
  Props.AddFilterOrValidate('Email', TSynValidateEmail.Create);
end;

property Aggregate: TClass read fAggregate.ItemClass;
The DDD's Entity class handled by this factory
- may be any TPersistent, but very likely a TSynAutoCreateFields class

property AggregateClass: string read GetAggregateName;
The DDD's Entity class name handled by this factory

property FieldMapping: TSQLRecordPropertiesMapping read fPropsMapping;
Access to the Aggregate / ORM field mapping

property Owner: TDDDRepositoryRestManager read fOwner;
The home repository owning this factory

property Props: TSQLPropInfoList read fAggregateRTTI;
The mapped DDD's Entity class published properties RTTI
property Repository: TInterfaceFactory read fInterface;
    The associated I*Query / I*Command repository interface

property Rest: TSQLRest read fRest;
    The associated TSQLRest instance

property Table: TSQLRecordClass read fTable;
    The ORM's TSQLRecord used for actual storage

property TableClass: string read GetTableName;
    The ORM's TSQLRecord class name used for actual storage

TDDDRepositoryRestQuery = class(TCQRSService)
    Abstract repository class to implement I*Query interface using RESTful ORM
    - actual repository implementation will just call the ORM*() protected method from the published
      Aggregate-oriented CQRS service interface

constructor Create(aFactory: TDDDRepositoryRestFactory); reintroduce; virtual;
    You should not have to use this constructor, since the instances would be injected by
    TDDDRepositoryRestFactory.TryResolve()

destructor Destroy; override;
    Finalize the used memory

function GetCount: integer; virtual;
    Return the number all currently selected aggregates
    - returns 0 if no select was available, 1 if it was a ORMGetSelectOne(), or the number of items
      after a ORMGetSelectAll()
    - this is a generic operation which would work for any class
    - if you do not need this method, just do not declare it in I*Command

class function GetRest(const Service: ICQRSService): TSQLRest;
    Returns the associated TSQLRest instance used in the associated factory
    - this method is able to extract it from a I*Query/I*Command instance, if it is implemented by a
      TDDDRepositoryRestQuery class
    - returns nil if the supplied Service is not recognized

property CurrentORMInstance: TSQLRecord read fCurrentORMInstance;
    Access to the current state of the underlying mapped TSQLRecord
    - is nil if no query was run yet
    - contains the queried object after a successful Select*() method
    - is either a single object, or a list of objects, via its internal CurrentORMInstance.FillTable cursor

property Factory: TDDDRepositoryRestFactory read fFactory;
    Access to the associated factory

TDDDRepositoryRestCommand = class(TDDDRepositoryRestQuery)
    Abstract class to implement I*Command interface using ORM's TSQLRecord
    - it will use an internal TSQLRestBatch for dual-phase commit, therefore implementing a generic
      Unit Of Work / Transaction pattern
**constructor** Create(aFactory: TDDDRepositoryRestFactory); override;

*This constructor will set default fBatch options*

**destructor** Destroy; override;

*Finalize the Unit Of Work context*
- any uncommitted change will be lost

**function** Commit: TCQRSResult; virtual;

*Write all pending changes prepared by Add/Update/Delete methods*
- this is the only mandatory method, to be declared in your I*Command*
- in practice, will send the current internal BATCH to the REST instance

**function** Delete: TCQRSResult; virtual;

*Perform a deletion on the currently selected aggregate*
- this is a generic operation which would work for any class
- if you do not need this method, just do not declare it in I*Command

**function** DeleteAll: TCQRSResult; virtual;

*Perform a deletion on all currently selected aggregates*
- this is a generic operation which would work for any class
- if you do not need this method, just do not declare it in I*Command

**function** Rollback: TCQRSResult; virtual;

*Flush any pending changes prepared by Add/Update/Delete methods*
- if you do not need this method, just do not publish it in I*Command*
- the easiest to perform a roll-back would be to release the I*Command instance - but you may explicitly reset the pending changes by calling this method
- in practice, will release the internal BATCH instance

**property** Batch: TSQLRestBatch read fBatch;

*Access to the low-level BATCH instance, used for dual-phase commit*
- you should not need to access it directly, but rely on Commit and Rollback methods to

TCQRSQueryObjectRest = class(TCQRSService)

*Abstract CQRS class tied to a TSQLRest instance for low-level persistence*
- not used directly by the DDD repositories (since they will rely on a TDDDRepositoryRestFactory for the actual ORM process), but may be the root class for any Rest-based infrastructure cross-cutting features

**constructor** Create(aRest: TSQLRest); reintroduce; virtual;

*Reintroduced constructor, allowing to specify the associated REST instance*

**constructor** CreateInjected(aRest: TSQLRest; const aStubsByGUID: array of TGUID; const aOtherResolvers: array of TInterfaceResolver; const aDependencies: array of TInterfacedObject); reintroduce;

*Reintroduced constructor, associating a REST instance with the supplied IoC resolvers (may be stubs/mocks, resolver classes or single instances)*

**constructor** CreateWithResolver(aRest: TSQLRest; aResolver: TInterfaceResolver; aRaiseEServiceExceptionIfNotFound: boolean=true); reintroduce; overload;

*Reintroduced constructor, associating a REST instance with the supplied IoC resolvers*
constructor CreateWithResolver(aResolver: TInterfaceResolver; aRaiseEServiceExceptionIfNotFound: boolean); overload; override;

This constructor would identify a TServiceContainer SOA resolver and set the Rest property - when called e.g. by TServiceFactoryServer.CreateInstance()

property Rest: TSQLEst read FRest;
Access to the associated REST instance

TDDDMonitoredDaemonProcess = class(TThread)
Abstract process thread class with monitoring abilities

constructor Create(aDaemon: TDDDMonitoredDaemon; aIndexInDaemon: integer); virtual;
Initialize the process thread for a given Service/Daemon instance

destructor Destroy; override;
Finalize the process thread

property IdleDelay: cardinal read fProcessIdleDelay;
Milliseconds delay defined before getting the next pending tasks
- equals TDDDMonitoredDaemon.ProcessIdleDelay, unless a fatal exception occurred during TDDDMonitoredDaemonProcess.ExecuteIdle method: in this case, the delay would been increased to 500 ms

TDDDMonitoredDaemonProcessRest = class(TDDDMonitoredDaemonProcess)
Abstract process thread class with monitoring abilities, using the ORM for pending tasks persistence
- a protected TSQLRecord instance will be maintained to store the processing task and its current state

TDDDMonitoredDaemon = class(TCQRSQueryObjectRest)
Abstract class using several process threads and with monitoring abilities
- able to implement any DDD Daemon/Service, with proper statistics gathering
- each TDDDMonitoredDaemon will own its TDDDMonitoredDaemonProcess

constructor Create(aRest: TSQLEst; aProcessThreadCount: integer); reintroduce; overload;
You should override this constructor to set the actual process
- i.e. define the fProcessClass protected property

constructor Create(aRest: TSQLEst); overload; override;
Abstract constructor, which should not be called by itself

destructor Destroy; override;
Finalize the Daemon

function RetrieveState(out Status: variant): TCQRSResult;
Monitor the Daemon/Service by returning some information as a TDocVariant
- its Status.stats sub object will contain global processing statistics, and Status.threadstats similar information, detailed by running thread
function Start: TCQRSResult; virtual;
  Launch all processing threads
  - any previous running threads are first stopped

function Stop(out Information: variant): TCQRSResult; virtual;
  Finalize all processing threads
  - and returns updated statistics as a TDocVariant

property ProcessIdleDelay: integer read fProcessIdleDelay write fProcessIdleDelay;
  How many milliseconds each process thread should wait before checking for pending tasks
  - default value is 50 ms, which seems good enough in practice

property ProcessThreadCount: integer read fProcessThreadCount;
  How many process threads should be created by this Daemon/Service

TDDDAdministratedDaemon = class(TCQRSService)
  Abstract class to implement an administrable service/daemon
  - a single administration daemon (running e.g. as a Windows Service) would be able to launch and
  administrate such process, via a remote REST link
  - inherited class should override the Internal* virtual abstract protected methods to supply the
  actual process (e.g. set a background thread)

constructor Create(const aUserName, aHashedPassword: RawUTF8; const aRoot: RawUTF8='admin'; const aServerNamedPipe: TFileName=''); reintroduce; overload;
  Initialize the administrable service/daemon with its own TSQLRestServer
  - will initialize and own its dedicated TSQLRestServerFullMemory
  - if aUserName is specified, authentication will be enabled, and a single TSQLAuthUser will be
    created, with the supplied credentials (the password matching
    TSQLAuthUser.PasswordHashHexa expectations)
  - under Windows, you can export the administration server as named pipe, if the optional
    aServerNamedPipe parameter is set

constructor Create(aAdministrationServer: TSQLRestServer); reintroduce; overload;
  virtual;
  Initialize the administrable service/daemon
  - aAdministrationServer.ServiceDefine(IAdministratedDaemon) will be called to publish the
    needed methods over it, to allow remote administration from a single administration daemon
    (installed e.g. as a Windows Service)
  - this constructor won't start the associated process, which would be idle until the Start method
    is called

destructor Destroy; override;
  Finalize the service/daemon
  - will call Halt() if the associated process is still running

function DaemonName: RawUTF8; virtual;
  Returns the daemon name
  - e.g. TMyOwnDaemon would return 'MyOwn' text
function DatabaseExecute(const DatabaseName, SQL: RawUTF8): TServiceCustomAnswer; virtual;
   IAdministratedDaemon command to execute a SQL query on an internal database
   - you may override this method to implement addition "pseudo-SQL" commands

function DatabaseList: TRawUTF8DynArray; virtual;
   IAdministratedDaemon command to retrieve all internal databases names
   - will return fInternalDatabases[].Model.Root values

function DatabaseTables(const DatabaseName: RawUTF8): TRawUTF8DynArray; virtual;
   IAdministratedDaemon command to return the table names of an internal database

function Halt(out Information: variant): TCQRSResult; virtual;
   IAdministratedDaemon command to Stop the associated process, then quit the executable
   - returning the same output information than Stop()

function RetrieveState(out Status: variant): TCQRSResult; virtual;
   Monitor the Daemon/Service by returning some information as a TDocVariant

function Start: TCQRSResult; virtual;
   IAdministratedDaemon command to launch the associated process
   - if the process was already running, returns cqrsAlreadyExists

function Stop(out Information: variant): TCQRSResult; virtual;
   IAdministratedDaemon command to finalize the associated process
   - and returns updated statistics as a TDocVariant

procedure CallbackReleased(const callback: IInvokable; const interfaceName: RawUTF8);
   IAdministratedDaemon command called when a callback is released on the client side

procedure Execute(RemotelyAdministrated: boolean);
   Run the daemon, until it is halted
   - if RemotelyAdministrated is FALSE, it will Start the process, then wait until the [Enter] key is
     pressed (to be used in pure console mode)
   - if RemotelyAdministrated is TRUE, it will follow remote activation from its administration
     server
   - both modes will log some minimal message on the console (if any)

procedure SubscribeLog(const Levels: TSynLogInfos; const Callback: ISynLogCallback;
   ReceiveExistingKB: cardinal); virtual;
   IAdministratedDaemon command to subscribe to a set of events for real-time remote monitoring
   of the specified log events

procedure WaitUntilHalted; virtual;
   This method will wait until Halt() is executed
   - i.e. protected fFinished TEvent is notified

property AdministrationHTTPServer: TObject read fAdministrationHTTPServer write
   fAdministrationHTTPServer;
   Reference to the WebSockets/HTTP server publishing AdministrationServer
   - is defined as an opaque TObject instance, to avoid unneeded dependencies
property AdministrationServer: TSQLRestServer read fAdministrationServer;
  Reference to the REST server publishing IAdministratedDaemon service
  - e.g. from named pipe local communication on Windows

property InternalSettings: TObject read fInternalSettings write SetInternalSettings;
  Access to the associated internal settings
  - is defined as an opaque TObject instance, to avoid unneeded dependencies

property InternalSettingsFolder: TFileName read fInternalSettingsFolder write fInternalSettingsFolder;
  Access to the associated internal settings storage folder

property Log: TSynLogFamily read fLog;
  Access to the associated logging class

property Status: TDDDAdministratedDaemonStatus read fStatus;
  The current status of the service/daemon

TDDDAdministratedThreadDaemon = class(TDDDAdministratedDaemon)
  Abstract class to implement an TThread-based administrable service/daemon
  - inherited class should override InternalStart and InternalRetrieveState abstract methods, and set
  the protected fThread with the processing thread

TDDDAdministratedRestDaemon = class(TDDDAdministratedDaemon)
  Abstract class to implement a TSQLRest-based administrable service/daemon
  - inherited class should override InternalStart and InternalRetrieveState abstract methods, and set
  the protected fRest with the processing TSQLRest

property Rest: TSQLRestServer read fRest;
  Read-only access to the associated REST instance
  - is assigned only between daemon Start/Stop

TDDDAdministratedDaemonMonitor = class(TSynAutoCreateFields)
  Abstract class to monitor an administrable service/daemon
  - including Input/Output statistics and connected Clients count
  - including OS Memory information

property Server: TSynMonitorServer read fServer;
  Information about the REST server process

property SystemMemory: variant read GetMemory;
  Information about the main System memory, as returned by the OS

TDDDApplication = class(TInjectableAutoCreateFields)
  Abstract class for implementing an Application Layer service
  - is defined as an TInjectableAutoCreateFields, so that any published properties defined as
  interfaces would be resolved at creation, and published properties defined as
  TPersistent/TSynPersistent will be managed by this instance, i.e. created and released with it

Types implemented in the mORMotDDD unit
TCQRSQueryAction = ( qaNone, qaSelect, qaGet, qaCommandDirect, qaCommandOnSelect, qaCommit );

*Which kind of process is about to take place after an CqrsBeginMethod()*

TCQRSQueryActions = set of TCQRSQueryAction;

*Define one or several process to take place after an CqrsBeginMethod()*

TCQRSQueryState = ( qsNone, qsQuery, qsCommand );

*The current step of a TCQRSQuery state machine*
- basic state is defined by the methods execution:
  - qsNone refers to the default state, with no currently selected values, nor any pending write request
  - qsQuery corresponds to a successful I*Query.Select*(), expecting either a I*Query.Get*(), or a I*Command.Add/Update/Delete
  - qsCommand corresponds to a successful I*Command.Add/Update/Delete, expected a I*Command.Commit

TCQRSResult = ( cqrsSuccess, cqrsSuccessWithMoreData, cqrsUnspecifiedError, cqrsBadRequest, cqrsNoMatch, cqrsNoMoreData, cqrsDataLayerError, cqrsInvalidCallback, cqrsInternalError, cqrsDDDValidationFailed, cqrsInvalidContent, cqrsAlreadyExists, cqrsNoPriorQuery, cqrsNoPriorCommand, cqrsNoMatch, cqrsNotImplemented, cqrsBusy, cqrsTimeout );

*Result enumerate for I*Query/I*Command CQRS repository service methods*
- cqrsSuccess will map the default TInterfaceStub returned value
- cqrsSuccessWithMoreData would be used e.g. for versioned publish/subscribe to notify the caller that there are still data available, and the call should be reiterated until cqrsSuccess is returned
- cqrsBadRequest would indicate that the method was not called in the expected workflow sequence
- cqrsNoMatch appear after a I*Query SelectBy*() method with no match
- cqrsNoMoreData indicates a GetNext*() method has no more matching data
- cqrsDataLayerError indicates a low-level error at database level
- cqrsInvalidCallback is returned if a callback is required for this method
- cqrsInternalError for an unexpected issue, like an Exception raised
- cqrsDDDValidationFailed will be triggered when
- cqrsInvalidContent for any I*Command method with invalid aggregate input value (e.g. a missing field)
- cqrsAlreadyExists for a I*Command.Add method with a primay key conflict
- cqrsNoPriorQuery for a I*Command.Update/Delete method with no prior call to SelectBy*()
- cqrsNoPriorCommand for a I*Command.Commit with no prior Add/Update/Delete
- cqrsNoMatch may be returned when there is no code yet for a method
- cqrsBusy is returned if the command could not be executed, since it is currently processing a request
- cqrsTimeout indicates that the method didn't succeed in the expected time
- otherwise, cqrsUnspecifiedError will be used for any other kind of error

TCQRSServiceClass = class of TCQRSService;

*Class-reference type (metaclass) of TCQRSService*

TDDDAdministratedDaemonClass = class of TDDDAdministratedDaemon;

*Type used to define a class kind of TDDDAdministratedDaemon*

TDDDAdministratedDaemonStatus = ( dsUnknown, dsCreated, dsStarted, dsStopped, dsHalted );

*Current status of an administrable service/daemon*
TDDDMonitoredDaemonProcessClass = class of TDDDMonitoredDaemonProcess;

Class-reference type (metaclass) to determine which actual thread class will implement the monitored process

TDDDMonitoredDaemonProcessState = ( dpsPending, dpsProcessing, dpsProcessed, dpsFailed );

The current state of a process thread

TDDRepositoryRestClass = class of TDDRepositoryRestQuery;

Class-reference type (metaclass) to implement I*Query or I*Command interface definitions using our RESTful ORM

TDDRepositoryRestFactoryObjArray = array of TDDRepositoryRestFactory;

Store reference of several factories, each with one mapping definition

Constants implemented in the mORMotDDD unit

CQRSRESULT_SUCCESS = [ cqrsSuccess, cqrsSuccessWithMoreData, cqrsNoMoreData, cqrsNotFound ];

Successfull result enumerates for I*Query/I*Command CQRS
- those items would generate no log entry
- i.e. any command not included in CQRSRESULT_SUCCESS nor CQRSRESULT_WARNING would trigger a sllDDDError log entry

CQRSRESULT_WARNING = [ cqrsNotFound, cqrsNoMatch ];

Dubious result enumerates for I*Query/I*Command CQRS
- those items would generate a sllDDDInfo log entry
- i.e. any command not included in CQRSRESULT_SUCCESS nor CQRSRESULT_WARNING would trigger a sllDDDError log entry

Functions or procedures implemented in the mORMotDDD unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToText</td>
<td>Returns the text equivalency of a CQRS result enumeration</td>
<td>2301</td>
</tr>
<tr>
<td>ToText</td>
<td>Returns the text equivalency of a CQRS state enumeration</td>
<td>2301</td>
</tr>
</tbody>
</table>

function ToText(res: TCQRSQueryState): PShortString; overload;
Returns the text equivalency of a CQRS state enumeration

function ToText(res: TCQRSResult): PShortString; overload;
Returns the text equivalency of a CQRS result enumeration
27.55. mORMotFastCgiServer.pas unit

Purpose: FastCGI HTTP/1.1 Server implementation for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the mORMotFastCgiServer unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynCrtSock</td>
<td>Classes implementing TCP/UDP/HTTP client and server protocol</td>
<td>1083</td>
</tr>
<tr>
<td>SynWinSock</td>
<td>Low level access to network Sockets for the Win32 platform</td>
<td>1843</td>
</tr>
</tbody>
</table>

Objects implemented in the mORMotFastCgiServer unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFastCGIServer</td>
<td>Handle Fast CGI</td>
<td>2303</td>
</tr>
<tr>
<td>TFCGIBeginRequest</td>
<td>RtBeginRequest record</td>
<td>2303</td>
</tr>
<tr>
<td>TFCGIHeader</td>
<td>FastCGI header</td>
<td>2302</td>
</tr>
</tbody>
</table>

TFCGIHeader = packed record

FastCGI header

Filler: byte;

Pad field

ID: word;

Identifies the FastCGI request to which the record belongs
- equals zero for a management request request
- non zero for an application record Used also to determine if the session is being multiplexed
Len: word;
    *FastCGI record length*
    - will send up to 64 KB of data per block

PadLen: byte;
    *Pad length to complete 8 bytes alignment boundary in FastCGI protocol*

RecType: TFCGIRecType;
    *FastCGI record type*

Version: byte;
    *FastCGI protocol version, ever constant 1*

TFCGIBeginRequest = packed record
    RtBeginRequest record

Filler: byte;
    *Pad field*

Filler2: array[1..5] of byte;
    *Pad field*

Header: TFCGIHeader;
    *FastCGI header*

KeepConn: boolean;
    *Keep connection*

Role: TFCGIRole;
    *FastCGI role*

TFastCGIServer = class(TObject)
    Handle Fast CGI
    - implements the official Fast CGI Specification available at
      @http://www.fastcgi.com/devkit/doc/fcgi-spec.html
    - this base type has virtual public methods ReadPacked and SendPacket, implementing the named
      pipe or socket defined by the single file descriptor sent by the web server which can be overridden
      by its children for proper socket/pipe handling

constructor Create(aServer: TSQLRestServer);
    *Create the object instance to run with the specified RESTful Server*

destructor Destroy; override;
    *Release the associated memory and handles*

function ReadPacked: RawUTF8; virtual;
    *Virtual method used to read a packet from the remote server*
    - must return '' on error
    - by default, use the single file descriptor sent by the web server, and expect to read data from
      the corresponding named pipe or TCP/IP socket
**function Run**: boolean; **virtual**;

The main loop of the FastCGI application
- loop until application is terminated
- use the associated RESTful Server to calculate the answer
- call the virtual methods ReadPacked and SendPacket to handle the response
- the FastCGI server must have been successfully connected before calling it
- return true if communication was made successfully

**function SendPacket**: (Buffer: pointer; BufferLen: integer): boolean; **virtual**;

Virtual method used to send a packed to the remote server
- must return FALSE on error
- by default, use the single file descriptor sent by the web server, and expect to write data to the corresponding named pipe or TCP/IP socket

**procedure LogOut**: **virtual**;

Method triggered when the Web server wants to abort the request
- do nothing by default - only to be implemented for Multiplex connection which are not enabled with this class

**procedure ProcessRequest**: **virtual**;

Method triggered to calculate the response
- expect fRequestHeaders, fRequestMethod, fRequestBody and fRequestURL properties as input
- update fResponseHeaders and fResponseContent properties as output

**property Server**: TSQLRestServer **read** fServer;

Associated RESTful Server

**Types implemented in the mORMotFastCgiServer unit**

TFCGILevelType = ( ltUnused, ltSocketSync, ltPipeSync );

FastCGI connection modes

TFCGIProtocolStatus = ( psRequestComplete, psCantMultiplexConnections, psOverloaded, psUnknownRole );

FastCGI level status (and error) code for END_REQUEST record

TFCGIRecType = ( rtBeginRequest, rtAbortRequest, rtEndRequest, rtParams, rtStdIn, rtStdOut, rtStdErr, rtData, rtGetValues, rtGetValuesResult, rtUnknown );

FastCGI record types, i.e. the general function that the record performs

TFCGIRole = ( rUnknown, rResponder, rAuthorizer, rFilter );

FastCGI roles, only Responder role is supported in this unit version

**Functions or procedures implemented in the mORMotFastCgiServer unit**

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMotFastCgIMainProc</td>
<td>The pascal version use our Synopse socket publishes a HTTP/1.1 RESTFUL JSON mORMot Server, using FASTCGI</td>
<td>2305</td>
</tr>
</tbody>
</table>
procedure mORMotFastCGIMainProc(Server: TSQLRestServer);

The pascal version use our Synops socket publishes a HTTP/1.1 RESTFUL JSON mORMot Server, using FASTCGI
- will raise an exception if the executable was not used as a FastCGI process, but a normal CGI process
- call this procedure in your main block of your program: it is up to the HTTP server to implement the request handling
27.56. mORMotHttpClient.pas unit

*Purpose:* HTTP/1.1 RESTful JSON Client classes for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The **mORMotHttpClient** unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.1.1.2.4</td>
<td>HTTP/1.1 protocol</td>
<td>2545</td>
</tr>
</tbody>
</table>

Units used in the **mORMotHttpClient** unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| mORMot | Common ORM and SOA classes for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1899 |
| SynBidirSock | Implements bidirectional client and server protocol, e.g. WebSockets
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 680 |
| SynCommons | Common functions used by most Synopse projects
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 717 |
| SynCrtSock | Classes implementing TCP/UDP/HTTP client and server protocol
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1083 |
| SynCrypto | Fast cryptographic routines (hashing and cypher)
- implements AES,XOR,ADLER32,MDS,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT
- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1139 |
| SynLog | Logging functions used by Synopse projects
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1363 |
| SynLZ | SynLZ Compression routines
- licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1393 |
| SynTable | Filter/database/cache/buffer/security/search/multithread/OS features
- as a complement to SynCommons, which tended to increase too much
- licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1721 |
| SynZip | Low-level access to ZLib compression (1.2.5 engine version)
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1845 |
**mORMotHttpClient class hierarchy**

**Objects implemented in the mORMotHttpClient unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLHttpClientGeneric</td>
<td>Abstract HTTP/1.1 RESTful JSON mORMot Client class</td>
<td>2307</td>
</tr>
<tr>
<td>TSQLHttpClientRequest</td>
<td>HTTP/1.1 RESTful JSON mORMot Client abstract class using either WinINet, WinHTTP or libcurl API</td>
<td>2310</td>
</tr>
<tr>
<td>TSQLHttpClientWebsockets</td>
<td>HTTP/1.1 RESTful JSON mORMot Client able to upgrade to WebSockets</td>
<td>2309</td>
</tr>
<tr>
<td>TSQLHttpClientWinHTTP</td>
<td>HTTP/1.1 RESTful JSON Client class using WinHTTP API</td>
<td>2311</td>
</tr>
<tr>
<td>TSQLHttpClientWinINet</td>
<td>HTTP/1.1 RESTful JSON mORMot Client class using WinINet API</td>
<td>2311</td>
</tr>
<tr>
<td>TSQLHttpClientWinSock</td>
<td>HTTP/1.1 RESTful JSON mORMot Client class using SynCrtSock's Sockets</td>
<td>2309</td>
</tr>
</tbody>
</table>

**TSQLHttpClientGeneric = class(TSQLRestClientURI)**

Abstract HTTP/1.1 RESTful JSON mORMot Client class
- this class, and other inherited classes defined in this unit, are thread-safe, since each of their URI() method is protected by a giant lock

```cpp
class TSQLHttpClientGeneric = class(TSQLRestClientURI)

constructor Create(const aServer: TSQLRestServerURIString; aModel: TSQLModel; aDefaultPort: integer; ahttps: boolean=false); reintroduce; overload;

  Connect to TSQLHttpServer via 'address:port/root' URI format
  - if port is not specified, aDefaultPort is used
  - if root is not specified, aModel.Root is used

constructor Create(const aServer, aPort: AnsiString; aModel: TSQLModel; ahttps: boolean=false; const aProxyName: AnsiString=''; const aProxyByPass: AnsiString=''; aSendTimeout: DWORD=0; aReceiveTimeout: DWORD=0; aConnectTimeout: DWORD=0); reintroduce; overload; virtual;

  Connect to TSQLHttpServer on aServer:aPort
  - optional aProxyName may contain the name of the proxy server to use, and aProxyByPass an optional semicolon delimited list of host names or IP addresses, or both, that should not be routed through the proxy - note that proxy parameters are currently not available for TSQLHttpClientWinSock
  - you can customize the default client timeouts by setting appropriate ConnectTimeout, SendTimeout and ReceiveTimeout parameters (in ms) - if you left the 0 default parameters, it would use global HTTP_DEFAULT_CONNECTTIMEOUT, HTTP_DEFAULT_SENDTIMEOUT and HTTP_DEFAULT_RECEIVETIMEOUT variable values
```

TSQLHttpClientGeneric = class(TSQLRestClientURI)

Abstract HTTP/1.1 RESTful JSON mORMot Client class
- this class, and other inherited classes defined in this unit, are thread-safe, since each of their URI() method is protected by a giant lock

**constructor** Create(const aServer: TSQLRestServerURIString; aModel: TSQLModel; aDefaultPort: integer; ahttps: boolean=false); reintroduce; overload;

  Connect to TSQLHttpServer via 'address:port/root' URI format
  - if port is not specified, aDefaultPort is used
  - if root is not specified, aModel.Root is used

**constructor** Create(const aServer, aPort: AnsiString; aModel: TSQLModel; ahttps: boolean=false; const aProxyName: AnsiString=''; const aProxyByPass: AnsiString=''; aSendTimeout: DWORD=0; aReceiveTimeout: DWORD=0; aConnectTimeout: DWORD=0); reintroduce; overload; virtual;

  Connect to TSQLHttpServer on aServer:aPort
  - optional aProxyName may contain the name of the proxy server to use, and aProxyByPass an optional semicolon delimited list of host names or IP addresses, or both, that should not be routed through the proxy - note that proxy parameters are currently not available for TSQLHttpClientWinSock
  - you can customize the default client timeouts by setting appropriate ConnectTimeout, SendTimeout and ReceiveTimeout parameters (in ms) - if you left the 0 default parameters, it would use global HTTP_DEFAULT_CONNECTTIMEOUT, HTTP_DEFAULT_SENDTIMEOUT and HTTP_DEFAULT_RECEIVETIMEOUT variable values
constructor CreateForRemoteLogging(const aServer: AnsiString; aLogClass: TSynLogClass; aPort: Integer=8091; const aRoot: RawUTF8='LogService');

Connect to a LogView HTTP Server for remote logging
- will associate the EchoCustom callback of the log class to this server
- the aLogClass.Family will manage this TSQLHttpGeneric instance life time, until application is closed or Family.EchoRemoteStop is called

function HostName: AnsiString;
  Returns 'Server:Port' current value

procedure DefinitionTo(Definition: TSynConnectionDefinition); override;
  Save the TSQLHttpClientGeneric properties into a persistent storage object
- CreateFrom() will expect Definition.ServerName to store the URI as 'server:port' or 'https://server:port', Definition.User/Password to store the TSQLRestClientURI.SetUser() information, and Definition.DatabaseName to store the extended options as an URL-encoded string

property Compression: TSQLHttpCompressions read fCompression write SetCompression;
  The compression algorithms usable with this client
  - equals [hcSynLZ] by default, since our SynLZ algorithm provides a good compression, with very low CPU use on server side
  - you may include hcDeflate, which will have a better compression ratio, be recognized by all browsers and libraries, but would consume much more CPU resources than hcSynLZ
  - if you include hcSynShaAes, it will use SHA-256/AES-256-CFB to encrypt the content (after SynLZ compression), if it is enabled on the server side:
    MyServer := TSQLHttpServer.Create('888',DataBase,'+',useHttpApi,32,secSynShaAes);
    - for fast and safe communication between stable mORMot nodes, consider using TSQLHttpClientWebSockets, leaving hcDeflate for AJAX or non mORMot clients, and hcSynLZ if you expect to have mORMot client(s)

property ConnectRetrySeconds: integer read fConnectRetrySeconds write fConnectRetrySeconds;
  How many seconds the client may try to connect after open socket failure
  - is disabled to 0 by default, but you may set some seconds here e.g. to let the server start properly, and let the client handle exceptions to wait and retry until the specified timeout is reached
  - this property is used only once at startup, then flushed to 0 once connected

property KeepAliveMS: cardinal read fKeepAliveMS write SetKeepAliveMS;
  The time (in milliseconds) to keep the connection alive with the TSQLHttpServer
  - default is 20000, i.e. 20 seconds

property Port: AnsiString read fPort;
  The Server IP port

property Server: AnsiString read fServer;
  The Server IP address

property UserAgent: SockString read fExtendedOptions.UserAgent write fExtendedOptions.UserAgent;
  Optional custom HTTP "User Agent:" header value
TSQLHttpClientWinSock = class(TSQLHttpClientGeneric)

HTTP/1.1 RESTful JSON mORMot Client class using SynCrtSock's Sockets
- will give the best performance on a local computer, but has been found out to be slower over a network
- is not able to use secure HTTPS protocol
- note that, in its current implementation, this class is not thread-safe: you need either to lock its access via a critical section, or initialize one client instance per thread

property Socket: THttpClientSocket read fSocket;
Internal HTTP/1.1 compatible client
- can be used e.g. to access SendTimeout and ReceiveTimeout properties

TSQLHttpClientWebsockets = class(TSQLHttpClientWinSock)

HTTP/1.1 RESTful JSON mORMot Client able to upgrade to WebSockets
- in addition to TSQLHttpClientWinSock, this client class is able to upgrade its HTTP connection to the WebSockets protocol, so that the server may be able to notify the client via a callback
- the internal Socket class will be in fact a THttpClientWebSockets instance, as defined in the SynBidirSock unit

constructor Create(const aServer, aPort: AnsiString; aModel: TSQLModel; aHttps: boolean=false; const aProxyName: AnsiString=''; const aProxyByPass: AnsiString=''; aSendTimeout: DWORD=0; aReceiveTimeout: DWORD=0; aConnectTimeout: DWORD=0);
override;

Connect to TSQLHttpServer on aServer:aPort
- this overridden method will handle properly WebSockets settings

function DefaultWebSocketProcessSettings: PWebSocketProcessSettings;
Returns a reference to default settings for every new WebSocket process

function WebSockets: THttpClientWebSockets;
Internal HTTP/1.1 and WebSockets compatible client
- you could use its properties after upgrading the connection to WebSockets

function WebSocketsConnect(const aWebSocketsEncryptionKey: RawUTF8; aWebSocketsAJAX: boolean=false; aWebSocketsCompression: boolean=true): RawUTF8;

Connect using a specified WebSockets protocol
- this method would call WebSocketsUpgrade, then ServerTimestampSynchronize
- it therefore expects SetUser() to have been previously called

function WebSocketsConnected: boolean;
Returns true if the connection is a running WebSockets
- may be false even if fSocket<>nil, e.g. when gracefully disconnected
function WebSocketsUpgrade(const aWebSocketsEncryptionKey: RawUTF8; aWebSocketsAJAX: boolean=false; aWebSocketsCompression: boolean=true): RawUTF8;
Upgrade the HTTP client connection to a specified WebSockets protocol
- The Model.Root URI will be used for upgrade
- if aWebSocketsAJAX equals default FALSE, it will use 'synopsebinary' i.e.
  TWebSocketProtocolBinaryProtocol, with AES-CFB 256 bits encryption if the encryption key text
  is not '' and optional SynLZ compression
- if aWebSocketsAJAX is TRUE, it will register the slower and less secure 'synopsejson' mode, i.e.
  TWebSocketProtocolJSON (to be used for AJAX debugging/test purposes only) and
  aWebSocketsEncryptionKey/aWebSocketsCompression parameters won't be used
- once upgraded, the client would automatically re-upgrade any new HTTP client link on
  automatic reconnection, so that use of this class should not be tied to a particular TCP/IP socket
- use OnWebSocketsUpgraded event to perform any needed initialization set, e.g. SOA real-time
  callbacks registration
- will return '' on success, or an error message on failure

procedure CallbackNonBlockingSetHeader(out Header: RawUTF8); override;
Will set the HTTP header as expected by THttpClientWebSockets.Request to perform the
Callback() query in wscNonBlockWithoutAnswer mode

property OnWebSocketsClosed: TNotifyEvent read fOnWebSocketsClosed write fOnWebSocketsClosed;
This event handler will be executed when the WebSocket link is destroyed
- may happen e.g. after graceful close from the server side, or after
  DisconnectAfterInvalidHeartbeatCount is reached

property OnWebSocketsUpgraded: TOnRestClientNotify read fOnWebSocketsUpgraded write fOnWebSocketsUpgraded;
This event will be executed just after the HTTP client has been upgraded to the expected
WebSockets protocol
- supplied Sender parameter will be this TSQLHttpClientWebsockets instance
- it will be executed the first time, and also on each reconnection occurring when the
  HTTP-TCP/IP link is re-created, and user re-authenticated
- this event handler is the right place to setup link-driven connection, e.g. SOA real-time
  callbacks registration (using Sender.Services)

property WebSocketLoopDelay: integer read fWebSocketLoopDelay write fWebSocketLoopDelay;
Customize the internal REST loop delay
- to be defined before WebSocketsUpdate/WebSocketsConnect
- will set TWebSocketProcessSettings.LoopDelay value at WebSocketsUpgrade
- will override LoopDelay from DefaultWebSocketProcessSettings

TSQLHttpClientRequest = class(TSQLHttpClientGeneric)
HTTP/1.1 RESTful JSON mORMot Client abstract class using either WinINet, WinHTTP or libcurl API
- not to be called directly, but via TSQLHttpClientWinINet or (even better)
  TSQLHttpClientWinHTTP overridden classes under Windows

Optional Password for Authentication
property AuthScheme: THttpRequestAuthentication read fExtendedOptions.Auth.Scheme
write fExtendedOptions.Auth.Scheme;

Optional Authentication Scheme

property AuthUserName: SynUnicode read fExtendedOptions.Auth.UserName write fExtendedOptions.Auth.UserName;

Optional User Name for Authentication

property IgnoreSSLCertificateErrors: boolean read fExtendedOptions.IgnoreSSLCertificateErrors
write fExtendedOptions.IgnoreSSLCertificateErrors;

Allows to ignore untrusted SSL certificates
- similar to adding a security exception for a domain in the browser

property Request: THttpRequest read fRequest;

Internal class instance used for the connection
- will return either a TWinInet, a TWinHTTP or a TCurlHTTP class instance

TSQLHttpClintWinINet = class(TSQLHttpClintRequest)

HTTP/1.1 RESTful JSON mORMot Client class using WinInet API
- this class is 15/20 times slower than TSQLHttpClint using SynCrtSock on a local machine, but
was found to be faster throughout local networks
- this class is able to connect via the secure HTTPS protocol
- it will retrieve by default the Internet Explorer proxy settings, and display some error messages
or authentication dialog on screen
- you can optionally specify manual Proxy settings at constructor level
- by design, the WinInet API should not be used from a service
- is implemented by creating a TWinInet internal class instance

TSQLHttpClintWinHTTP = class(TSQLHttpClintRequest)

HTTP/1.1 RESTful JSON Client class using WinHTTP API
- has a common behavior as THttpClintSocket() but seems to be faster over a network and is able
to retrieve the current proxy settings (if available) and handle secure HTTPS connection - so it
seems to be used in your client programs: TSQLHttpClint will therefore map to this class
- WinHTTP does not share directly any proxy settings with Internet Explorer. The default WinHTTP
proxy configuration is set by either proxycfg.exe on Windows XP and Windows Server 2003 or
earlier, either netsh.exe on Windows Vista and Windows Server 2008 or later; for instance, you
can run "proxycfg -u" or "netsh winhttp import proxy source=ie" to use the current user's proxy
settings for Internet Explorer (under 64 bit Vista/Seven, to configure applications using the 32 bit
WinHttp settings, call netsh or proxycfg bits from %SystemRoot%\SysWOW64 folder explicitly)
- you can optionally specify manual Proxy settings at constructor level
- by design, the WinHTTP API can be used from a service or a server
- is implemented by creating a TWinHTTP internal class instance

Types implemented in the mORMotHttpClient unit

TSQLHttpClint = TSQLHttpClintWinHTTP;

HTTP/1.1 RESTful JSON default mORMot Client class
- under Windows, maps the TSQLHttpClintWinHTTP class

Used for DI-2.1.1.2.4 (page 2545).

TSQLHttpClintRequestClass = class of TSQLHttpClintRequest;
Meta-class of TSQLHttpClientRequest types

TSQLHttpCompression = ( hcSynLZ, hcDeflate, hcSynShaAes );

For WebSockets for hcSynShaAes available compression algorithms for transmission
- SynLZ is faster than Deflate, but not standard: use hcSynLZ for Delphi clients, but hcDeflate for AJAX or any HTTP clients
- with hcSynLZ, the 440 KB JSON for TTestClientServerAccess._TSQLHttpClient is compressed into 106 KB with no speed penalty (it’s even a bit faster) whereas hcDeflate with its level set to 1 (fastest), is 25 % slower
- hcSynShaAes will use SHA-256/AES-256-CFB to encrypt the content (after SynLZ compression), via SynCrypto.CompressShaAes() function
- here hcDeflate will use in fact gzip content encoding, since deflate is inconsistent between browsers: http://stackoverflow.com/a/9186091
- TSQLHttpClientGeneric.Compression default property is [hcSynLZ]

TSQLHttpCompressions = set of TSQLHttpCompression;

Set of available compressions schemes

TSQLHttpsClient = TSQLHttpClientWinHTTP;

HTTP/HTTPS RESTful JSON default mORMot Client class
- under Windows, maps the TSQLHttpClientWinHTTP class, or TSQLHttpClientCurl under Linux

Variables implemented in the mORMotHttpClient unit

HttpClientFullWebSocketsLog: Boolean;
A global hook variable, able to set WebSockets logging to full verbose
- checked by TSQLHttpClientWebsockets.WebSocketsConnect()
27.57. mORMotHttpServer.pas unit

Purpose: HTTP/1.1 RESTFUL JSON Server classes for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The mORMotHttpServer unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.1.1.2.4</td>
<td>HTTP/1.1 protocol</td>
<td>2545</td>
</tr>
</tbody>
</table>

Units used in the mORMotHttpServer unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynBidirSock</td>
<td>Implements bidirectional client and server protocol, e.g. WebSockets</td>
<td>680</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrtSock</td>
<td>Classes implementing TCP/UDP/HTTP client and server protocol</td>
<td>1083</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
<tr>
<td></td>
<td>- implements AES,XOR,ADLER32,MDS,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLZ</td>
<td>SynLZ Compression routines</td>
<td>1393</td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynZip</td>
<td>Low-level access to ZLib compression (1.2.5 engine version)</td>
<td>1845</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>
mORMotHttpServer class hierarchy

Objects implemented in the *mORMotHttpServer* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EHttpServerException</td>
<td>For WebSockets for CompressShaAes() exception raised in case of a HTTP Server error</td>
<td>2314</td>
</tr>
<tr>
<td>TSQLHTTPRemoteLogServer</td>
<td>Limited HTTP server which is will receive remote log notifications</td>
<td>2320</td>
</tr>
<tr>
<td>TSQLHttpServer</td>
<td>HTTP/1.1 RESTFUL JSON mORMot Server class</td>
<td>2314</td>
</tr>
</tbody>
</table>

EHttpServerException = class(ECommunicationException)

*For WebSockets for CompressShaAes() exception raised in case of a HTTP Server error*

TSQLHttpServer = class(TSynPersistentLock)

**HTTP/1.1 RESTFUL JSON mORMot Server class**

- this server is multi-threaded and not blocking
- under Windows, it will first try to use fastest http.sys kernel-mode server (i.e. create a THttpApiServer instance); it should work OK under XP or WS 2K3 - but you need to have administrator rights under Vista or Seven: if http.sys fails to initialize, it will use the socket-based THttpServer; a solution is to call the THttpApiServer.AddUrlAuthorize class method during program setup for the desired port, or define a useHttpApiRegisteringURI kind of server, in order to allow it for every user
- under Linux, only THttpServer is available
- you can specify useBidirSocket kind of server (i.e. TWebSocketServerRest) if you want the HTTP protocol connection to be upgraded to a WebSockets mode, to allow immediate callbacks from the server to the client
- just create it and it will serve SQL statements as UTF-8 JSON
- for a true AJAX server, expanded data is prefered - your code may contain:
  
  ```
  DBServer.NoAJAXJSON := false;
  ```

*Used for DI-2.1.1.2.4 (page 2545).*
**constructor** Create(aServer: TSQLRestServer; aDefinition: TSQLHttpServerDefinition; aForcedKind: TSQLHttpServerOptions=HTTP_DEFAULT_MODE; aWebSocketsLoopDelay: integer=0); reintroduce; overload;

Create a Server instance, binded and listening on a TCP port to HTTP requests
- raise a EHttpServer exception if binding failed
- specify one TSQLRestServer instance to be published, and the associated transmission definition; other parameters would be the standard one
- only the supplied aDefinition.Authentication will be defined
- under Windows, will use http.sys with automatic URI registration, unless aDefinition.WebSocketPassword is set and binary WebSockets would be expected with the corresponding encryption, or aForcedKind is overridden
- optional aWebSocketsLoopDelay parameter could be set for tuning WebSockets responsiveness

Used for DI-2.1.1.2.4 (page 2545).

**constructor** Create(const aPort: AnsiString; aServer: TSQLRestServer; const aDomainName: AnsiString=’’; aHttpServerKind: TSQLHttpServerOptions=HTTP_DEFAULT_MODE; aRestAccessRights: PSQLAccessRights=nil; ServerThreadPoolCount: Integer=32; aHttpServerSecurity: TSQLHttpServerSecurity=secNone; const aAdditionalURL: AnsiString=''); const aQueueName: SynUnicode=''); reintroduce; overload;

Create a Server instance, binded and listening on a TCP port to HTTP requests
- raise a EHttpServer exception if binding failed
- specify one TSQLRestServer server class to be used
- port is an AnsiString, as expected by the WinSock API - in case of useHttpSocket or useBidirSocket kind of server, you can specify the public server address to bind to: e.g. '1.2.3.4:1234' - even for http.sys, the public address could be used e.g. for TSQLRestServer.SetPublicURI()
- aDomainName is the URLprefix to be used for HttpAddUrl API call
- the aHttpServerSecurity can be set to secSSL to initialize a HTTPS instance (after proper certificate installation as explained in the SAD pdf), or to secSynShaAes if you want our proprietary SHA-256 / AES-256-CTR encryption identified as "ACCEPT-ENCODING: synshaaes"
- optional aAdditionalURL parameter can be used e.g. to registry an URI to server static file content, by overriding TSQLHttpServer.Request
- for THttpApiServer, you can specify an optional name for the HTTP queue

Used for DI-2.1.1.2.4 (page 2545).
constructor Create(const aPort: AnsiString; const aServers: array of TSQLRestServer; 
const aDomainName: AnsiString=''; ahHttpServerKind: 
TSQlHttpServerOptions=HTTP_DEFAULT_MODE; ServerThreadPoolCount: Integer=32; 
ahHttpServerSecurity: TSQLHttpServerSecurity=secNone; const aAdditionalURL: 
AnsiString=''; const aQueueName: SynUnicode=''; aHeadersUnFiltered: 
boolean=false); reintroduce; overload;

Create a Server instance, binded and listening on a TCP port to HTTP requests
- raise a EHttpServer exception if binding failed
- specify one or more TSQLRestServer server class to be used: each class must have an unique
  Model.Root value, to identify which TSQLRestServer instance must handle a particular request
  from its URI
- port is an AnsiString, as expected by the WinSock API - in case of useHttpSocket or
  useBidirSocket kind of server, you should specify the public server address to bind to: e.g.
  '1.2.3.4:1234' - even for http.sys, the public address could be used e.g. for
  TSQLRestServer.SetPublicURI()
- aDomainName is the URLprefix to be used for HttpAddUrl API call: it could be either a fully
  qualified case-insensitive domain name an IPv4 or IPv6 literal string, or a wildcard ('+' will bound
  to all domain names for the specified port, '*' will accept the request when no other listening
  hostnames match the request for that port) - this parameter is ignored by the
  TSQlHttpApiServer instance
- ahHttpServerKind defines how the HTTP server itself will be implemented: it will use by default
  optimized kernel-based http.sys server (useHttpApi), optionally registering the URI
  (useHttpApiRegisteringURI) if needed, or using the standard Sockets library (useHttpSocket),
  possibly in its WebSockets-friendly version (useBidirSocket - you should call the
  WebSocketsEnable method to initialize the available protocols)
- by default, the PSQLAcessRights will be set to nil
- the ServerThreadPoolCount parameter will set the number of threads to be initialized to
  handle incoming connections (default is 32, which may be sufficient for most cases, maximum is
  256)
- the aHttpServerSecurity can be set to secSSL to initialize a HTTPS instance (after proper
  certificate installation as explained in the SAD pdf), or to secSynShaAes if you want our
  proprietary SHA-256 / AES-256-CTR encryption identified as "ACCEPT-ENCODING: synshaes"
- optional aAdditionalURL parameter can be used e.g. to registry an URI to server static file
  content, by overriding TSQLHttpServer.Request
- for THttpApiServer, you can specify an optional name for the HTTP queue
- for THttpServer, you can force aHeadersUnFiltered flag

Used for DI-2.1.1.2.4 (page 2545).

destructor Destroy; override;

Release all memory, internal mORMot server and HTTP handlers

Try to register another TSQLRestServer instance to the HTTP server
- each TSQLRestServer class must have an unique Model.Root value, to identify which instance must handle a particular request from its URI
- an optional aRestAccessRights parameter is available to override the default HTTP_DEFAULT_ACCESS_RIGHTS access right setting - but you shall better rely on the authentication feature included in the framework
- the aHttpServerSecurity can be set to secSSL to initialize a HTTPS instance (after proper certificate installation as explained in the SAD pdf), or to secSynShaAs if you want our proprietary SHA-256 / AES-256-CTR encryption identified as "ACCEPT-ENCODING: synshaas"
- return true on success, false on error (e.g. duplicated Root value)

Used for DI-2.1.1.2.4 (page 2545).

function DBServerFind(aServer: TSQLRestServer): integer;

Find the first instance of a registered REST server
- note that the same REST server may appear several times in this HTTP server instance, e.g. with diverse security options

function RemoveServer(aServer: TSQLRestServer): boolean;

Un-register a TSQLRestServer from the HTTP server
- each TSQLRestServer class must have an unique Model.Root value, to identify which instance must handle a particular request from its URI
- return true on success, false on error (e.g. specified server not found)

function WebSocketsEnable(aServer: TSQLRestServer; const aWebSocketsEncryptionKey: RawUTF8; aWebSocketsAJAX: boolean=false; aWebSocketsCompressed: boolean=true): TWebSocketServerRest; overload;

Defines the useBidirSocket WebSockets protocol to be used for a REST server
- same as the overloaded WebSocketsEnable() method, but the URI will be forced to match the aServer.Model.Root value, as expected on the client side by TSQLHttpClientWebsockets.WebSocketsUpgrade()

function WebSocketsEnable(const aWebSocketsURI, aWebSocketsEncryptionKey: RawUTF8; aWebSocketsAJAX: boolean=false; aWebSocketsCompressed: boolean=true): TWebSocketServerRest; overload;

Defines the WebSockets protocols to be used for useBidirSocket
- i.e. 'synopsebinary' and optionally 'synopsejson' protocols
- if aWebSocketsURI is '', any URI would potentially upgrade; you can specify an URI to limit the protocol upgrade to a single REST server
- TWebSocketProtocolBinary will always be registered by this method
- if the encryption key text is not '', TWebSocketProtocolBinary will use AES-CFB 256 bits encryption
- if aWebSocketsAJAX is TRUE, it will also register TWebSocketProtocolJSON so that AJAX applications would be able to connect to this server
- this method does nothing if the associated HttpServer class is not a TWebSocketServerRest (i.e. this instance was not created as useBidirSocket)
procedure DomainHostRedirect(const aDomain, aURI: RawUTF8);

Register a domain name to be redirected to a given Model.Root
- i.e. can be used to support some kind of virtual hosting
- by default, the URI would be used to identify which TSQLRestServer instance to use, and the incoming HOST value would just be ignored
- you can specify here domain names which would be checked against the incoming HOST header, to redirect to a given URI, as such:
  DomainHostRedirect('project1.com', 'root1');
  DomainHostRedirect('project2.com', 'root2');
  DomainHostRedirect('blog.project2.com', 'root2/blog');

for the last entry, you may have for instance initialized a MVC web server on the 'blog' sub-URI of the 'root2' TSQLRestServer via:

constructor TMyMCApplication.Create(aRestModel: TSQLRest; aInterface: PTypeInfo);
...
  fMainRunner := TMVCRunOnRestServer.Create(self, nil, 'blog');
  ...

- if aURI="" is given, the corresponding host redirection will be disabled
- note: by design, 'something.localhost' is likely to be not recognized as aDomain, since 'localhost' can not be part of proper DNS resolution

procedure RootRedirectToURI(const aRedirectedURI: RawUTF8; aRegisterURI: boolean=true; aHttps: boolean=false);

Allow to temporarily redirect ip:port root URI to a given sub-URI
- by default, only sub-URI, as defined by TSQLRestServer.Model.Root, are registered - you can define here a sub-URI to reach when the main server is directly accessed from a browser, e.g. localhost:port will redirect to localhost:port/RedirectedURI
- for http.sys server, would try to register '/' if aRegisterURI is TRUE
- by default, will redirect http://localhost:port unless you set aHttpServerSecurity=secSSL so that it would redirect https://localhost:port

procedure Shutdown(noRestServerShutdown: boolean=false);

You can call this method to prepare the HTTP server for shutting down
- it will call all associated TSQLRestServer.Shutdown methods, unless noRestServerShutdown is true
- note that Destroy won't call this method on its own, since the TSQLRestServer instances may have a life-time uncoupled from HTTP process

property AccessControlAllowCredential: boolean read fAccessControlAllowCredential write fAccessControlAllowCredential;

Enable cookies, authorization headers or TLS client certificates CORS exposition
- this option works with the AJAX XMLHttpRequest.withCredentials property on client/JavaScript side, as stated by
  @https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/withCredentials
- see
  @https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Creden
tials
property AccessControlAllowOrigin: RawUTF8 read fAccessControlAllowOrigin write SetAccessControlAllowOrigin;

Enable cross-origin resource sharing (CORS) for proper AJAX process
- see @https://developer.mozilla.org/en-US/docs/HTTP/Access_control_CORS
- can be set e.g. to '*' to allow requests from any site/domain; or specify an CSV white-list of URI to be allowed as origin e.g. as 'https://foo.example1,https://foo.example2' or 'https://*.foo.example' or (faster) '*.foo.example1,*.foo.example2' following the TMatch syntax
- see also AccessControlAllowCredential property

property DBServer[Index: integer]: TSQLRestServer read GetDBServer;

Read-only access to all internal servers

Used for DI-2.1.1.2.4 (page 2545).

property DBServerAccessRight[Index: integer]: PSQLAccessRights write SetDBServerAccessRight;

Write-only access to all internal servers access right
- can be used to override the default HTTP_DEFAULT_ACCESS_RIGHTS setting

property DBServerCount: integer read GetDBServerCount;

Read-only access to the number of registered internal servers

property DomainName: AnsiString read fDomainName;

The URL prefix used for internal HttpAddUrl API call

property HttpServer: THttpServerGeneric read fHttpServer;

The associated running HTTP server instance
- either THttpApiServer (available only under Windows), THttpServer or TWebSocketServerRest (on any system)

property OnlyJSONRequests: boolean read fOnlyJSONRequests write fOnlyJSONRequests;

Set this property to TRUE if the server must only respond to request of MIME type APPLICATION/JSON
- the default is false, in order to allow direct view of JSON from any browser

property Port: AnsiString read fPort;

The TCP/IP (address and) port on which this server is listening to
- may contain the public server address to bind to: e.g. '1.2.3.4:1234'
- see PublicAddress and PublicPort properties if you want to get the true IP port or address

property PublicAddress: RawUTF8 read fPublicAddress;

The TCP/IP public address on which this server is listening to
- equals e.g. '1.2.3.4' if Port = '1.2.3.4:1234'
- if Port does not contain an explicit address (e.g. '1234'), the current computer host name would be assigned as PublicAddress

property PublicPort: RawUTF8 read fPublicPort;

The TCP/IP public port on which this server is listening to
- equals e.g. '1234' if Port = '1.2.3.4:1234'
property RedirectServerRootUriForExactCase: boolean read fRedirectServerRootUriForExactCase write fRedirectServerRootUriForExactCase;

Enable redirectoin to fix any URI for a case-sensitive match of Model.Root
- by default, TSQLRestServer.Model.Root would be accepted with case insensitivity; but it may induce errors for HTTP cookies, since they are bound with '; Path=/ModelRoot', which is case-sensitive on the browser side
- set this property to TRUE so that only exact case URI would be handled by TSQLRestServer.URI(), and any case-sensitive URIs (e.g. /Root/... or /ROOT/...) would be temporary redirected to Model.Root (e.g. /root/...) via a HTTP 307 command

TSQLHTTPRemoteLogServer = class(TSQLHttpServer)

Limited HTTP server which is will receive remote log notifications
- this will create a simple in-memory mORMot server, which will trigger a supplied callback when a remote log is received
- see TSQLHttpClientWinGeneric.CreateForRemoteLogging() for the client side
- used e.g. by the LogView tool

constructor Create(const aRoot: RawUTF8; aPort: integer; const aEvent: TRemoteLogReceivedOne); reintroduce;

Initialize the HTTP server and an internal mORMot server
- you can share several HTTP log servers on the same port, if you use a dedicated root URI and use the http.sys server (which is the default)

destructor Destroy; override;

Release the HTTP server and its internal mORMot server

procedure RemoteLog(Ctxt: TSQLRestServerURIContext);

This HTTP server will publish a 'RemoteLog' method-based service
- expecting PUT with text as body, at http://server/root/RemoteLog

property Server: TSQLRestServerFullMemory read fServer;

The associated mORMot server instance running with this HTTP server

Types implemented in the mORMotHttpServer unit

TRemoteLogReceivedOne = procedure(const Text: RawUTF8) of object;

Callback expected by TSQLHTTPRemoteLogServer to notify about a received log

TSQLHttpServerOptions = ( useHttpApi, useHttpApiRegisteringURI, useHttpSocket, useBidirSocket );

Available running options for TSQLHttpServer.Create() constructor
- useHttpApi to use kernel-mode HTTP.SYS server (THttpApiServer) with an already registered URI (default way, similar to IIS/WCF security policy as specified by Microsoft) - you would need to register the URI by hand, e.g. in the Setup program, via code similar to this one:

  THttpApiServer.AddUrlAuthorize('root','888',false, '+')

- useHttpApiRegisteringURI will first registry the given URI, then use kernel-mode HTTP.SYS server (THttpApiServer) - will need Administrator execution rights at least one time (e.g. during setup); note that if the URI is already registered, the server will still be launched, even if the program does not run as Administrator - it is therefore sufficient to run such a program once as Administrator to register the URI, when this useHttpApiRegisteringURI option is set
- useHttpSocket will use the standard Sockets library (i.e. socket-based THttpServer) - it will trigger
the Windows firewall popup UAC window at first execution
- useBidirSocket will use the standard Sockets library but via the TWebSocketServerRest class, allowing HTTP connection upgrade to the WebSockets protocol, allowing immediate event callbacks in addition to the standard RESTful mode
- the first item should be the preferred one (see HTTP_DEFAULT_MODE)

```pascal
TSQLHttpServerSecurity = ( secNone, secSSL, secSynShaAes );
```

Available security options for TSQHttpServer.Create() constructor
- default secNone will use plain HTTP connection
- secSSL will use HTTPS secure connection
- secSynShaAes will use our proprietary SHA-256 / AES-256-CTR encryption identified as 'synshaaes' as ACCEPT-ENCODING: header parameter

**Constants implemented in the mORMotHttpServer unit**

```pascal
HTTP_DEFAULT_ACCESS_RIGHTS: PSQLAccessRights = @SUPERVISOR_ACCESS_RIGHTS;
```

The default access rights used by the HTTP server if none is specified

```pascal
HTTP_DEFAULT_MODE = useHttpApiRegisteringURI;
```

The kind of HTTP server to be used by default
- will define the best available server class, depending on the platform

**Variables implemented in the mORMotHttpServer unit**

```pascal
HttpServerFullWebSocketsLog: Boolean;
```

A global hook variable, able to enhance WebSockets logging
- when a TSQLHttpServer is created from a TSQLHttpServerDefinition
27.58. mORMoti18n.pas unit

*Purpose*: Internationalization (i18n) routines and classes for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The *mORMoti18n* unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.3.1.3</td>
<td>Automated i18n</td>
<td>2550</td>
</tr>
</tbody>
</table>

Units used in the *mORMoti18n* unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>mORMot</em></td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><em>SynCommons</em></td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

![mORMoti18n class hierarchy](image_url)

**mORMoti18n class hierarchy**

Objects implemented in the *mORMoti18n* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>TLanguage</em></td>
<td>A common record to identify a language</td>
<td>2322</td>
</tr>
<tr>
<td><em>TLanguageFile</em></td>
<td>Class to load and handle translation files (fr.msg, de.msg, ja.msg.. e.g.)</td>
<td>2324</td>
</tr>
</tbody>
</table>

**TLanguage = object(TObject)**

*A common record to identify a language*

*Used for DI-2.3.1.3 (page 2550).*

**CharSet**: integer;
*The corresponding Char Set*

**CodePage**: cardinal;
*The corresponding Code Page*

**Index**: TLanguages;
*As in LanguageAbr[index], LANGUAGE_NONE before first SetLanguageLocal()*
LCID: cardinal;

    The corresponding Windows LCID

function Abr: RawByteString;

    Returns two-chars long language abbreviation ('HE' e.g.)

function Name: string;

    Returns fully qualified language name ('Hebrew' e.g.), using current UI language
    - return "string" type, i.e. UnicodeString for Delphi 2009 and up

procedure Fill(Language: TLanguages);

    Initializes all TLanguage object fields for a specific language
TLanguageFile = class(TObject)

Class to load and handle translation files (fr.msg, de.msg, ja.msg.. e.g.)
- This standard .msg text file contains all the program resources translated into any language.
- Unicode characters (Chinese or Japanese) can be used.
- The most important part of this file is the [Messages] section, which contain all the text to be displayed in NumericValue=Text pairs. The numeric value is a hash (i.e. unique identifier) of the Text. To make a new translation, the "Text" part of these pairs must be translated, but the NumericValue must remain the same.
- In the "Text" part, translator must be aware of some important characters, which must NOT be modified, and appears in exactly the same place inside the translated text:
  1. | indicates a CR (carriage return) character;
  2. ~ indicates a LF (line feed) character;
  3. , sometimes is a comma inside a sentence, but is also used some other times as a delimiter between sentences;
  4. %s will be replaced by a textual value before display;
  5. %d will be replaced by a numerical value before display;
  some HTML code may appear (e.g. <br><font color="clnavy">...) and all text between < and > must NOT be modified;
  6. no line feed or word wrap is to be used inside the "Text" part; the whole NumericValue=Text pair must be contained in an unique line, even if it is huge.
- Some other sections appears before the [Messages] part, and does apply to windows as they are displayed on screen. By default, the text is replaced by a _ with a numerical value pointing to a text inside the [Messages] section. On some rare occasion, this default translation may be customized: in such cases, the exact new text to be displayed can be used instead of the _1928321 part. At the end of every line, the original text (never used, only put there for translator convenience) was added.
- In order to add a new language, the steps are to be performed:
  0. Extract all english message into a .txt ansi file, by calling the ExtractAllResources() procedure in the main program
  1. Use the latest .txt original file, containing the original English messages
  2. Open this file into a text editor (not Microsoft Word, but a real text editor, like the Windows notepad)
  3. Translate the English text into a new language; some Unicode characters may be used
  4. Save this new file, with the ISO two chars corresponding to the new language as file name, and .msg as file extension (e.g. FR.msg for French or RU.msg for Russian).
  5. By adding this .msg file into the PhD.exe folder, the PC User software will automatically find and use it to translate the User Interface on the fly. Each user is able to select its own preferred translation.
  6. The translator can perform the steps 3 to 5 more than once, to see in real time its modifications: he/she just has to restart the PC software to reload the updated translations.

Used for DI-2.3.1.3 (page 2550).

Language: TLanguage;

Identify the current language

constructor Create(aLanguageLocale: TLanguages); overload;

Load corresponding *.msg translation text file from the current exe directory

Used for DI-2.3.1.3 (page 2550).
constructor Create(const aFileName: TFileName; aLanguageLocale: TLanguages);
    Specify a text file containing the translation messages for a language
    Used for DI-2.3.1.3 (page 2550).
destructor Destroy; override;
    Free translation tables memory

function BooleanToString(Value: boolean): string;
    Convert the supplied boolean constant into ready to be displayed text
    - by default, returns 'No' for false, and 'Yes' for true
    - returns the text as generic string type, ready to be used in the VCL

function DateTimeToText(const DateTime: TDateTime): string; overload;
    Convert a date and time into a ready to be displayed text on the screen
    Used for DI-2.3.1.3 (page 2550).

function DateTimeToText(const Time: TTimeLog): string; overload;
    Convert a date and time into a ready to be displayed text on the screen
    Used for DI-2.3.1.3 (page 2550).

function DateTimeToText(const Time: TTimeLogBits): string; overload;
    Convert a date and time into a ready to be displayed text on the screen
    Used for DI-2.3.1.3 (page 2550).

function DateToText(const DateTime: TDateTime): string; overload;
    Convert a date into a ready to be displayed text on the screen
    Used for DI-2.3.1.3 (page 2550).

function DateToText(const Time: TTimeLog): string; overload;
    Convert a date into a ready to be displayed text on the screen
    Used for DI-2.3.1.3 (page 2550).

function DateToText(const Time: TTimeLogBits): string; overload;
    Convert a date into a ready to be displayed text on the screen
    Used for DI-2.3.1.3 (page 2550).
function PropToString(Prop: TSQLPropInfo; Instance: TSQLRecord; Client: TSQLRest): string;
Convert a TSQLRecord published property value into ready to be displayed text
- will convert any sftUTF8Text/sftAnsiText into ready to be displayed text
- will convert any sftInteger/sftFloat/sftCurrency into its textual value
- will convert any sftBoolean, sftEnumerate, sftDateTime, sftUnixTime or sftTimeLog/sftModTime/sftCreateTime into the corresponding text, depending on the current language
- will convert a sftSet property value to a list of all set enumerates, separated by #13#10
- will convert any sftID to 'Record Name', i.e. the value of the main property (mostly 'Name') of the referenced record
- will convert any sftRecord to 'Table Name: Record Name'
- will ignore sftBlob field
- returns the text as generic string type, ready to be used in the VCL

function ReadParam(const ParamName: RawUTF8): string;
Read a parameter, stored in the .msg file before any [Section]

function TimeToText(const DateTime: TDateTime): string; overload;
Convert a time into a ready to be displayed text on the screen
Used for DI-2.3.1.3 (page 2550).

function TimeToText(const Time: TTimeLogBits): string; overload;
Convert a time into a ready to be displayed text on the screen
Used for DI-2.3.1.3 (page 2550).

function TimeToText(const Time: TTimeLog): string; overload;
Convert a time into a ready to be displayed text on the screen
Used for DI-2.3.1.3 (page 2550).

procedure FormTranslateOne(aForm: TComponent);
Translate the english captions of a TForm into the current UI language
- must be called once with english captions
- call automatically if conditional USEFORMCREATEHOOK is defined

procedure LoadFromFile(const aFileName: TFileName);
Fill translation tables from text file containing the translation messages
- handle on the fly UTF-8 and UNICODE decode into the corresponding ANSI CHARSET, or into UnicodeString for Delphi 2009 and up (checking UTF-8 or Unicode BOM marker is available)

procedure Translate(var English: string);
Translate an English string into a localized string
- English is case-sensitive (same as standard gettext)
- translations are stored in Messages[] and Text properties
- expect parameter as generic VCL string (i.e. UnicodeString for Delphi 2009 and up)
Used for DI-2.3.1.3 (page 2550).

Types implemented in the mORMoti18n unit

TCompareFunction = function(const S1, S2: AnsiString): Integer;
Function prototype for comparing two Ansi strings
- used for comparison within the current selected language

TLanguages = ( lngHebrew, lngGreek, lngLatin, lngDari, lngBosnian, lngCatalan,
    lngCorsican, lngCzech, lngCoptic, lngSlavic, lngWelsh, lngDanish, lngGerman, lngArabic,
    lngEnglish, lngSpanish, lngFarsi, lngFinnish, lngFrench, lngIrish, lngGaelic,
    lngArmenian, lngBulgarian, lngHungarian, lngArmenian, lngIndonesian, lngInterlingue,
    lngIcelandic, lngItalian, lngJapanese, lngKorean, lngTibetan, lngLithuanian,
    lngMalagasy, lngNorwegian, lngOccitan, lngPortuguese, lngPolish, lngRomanian,
    lngRussian, lngSanskrit, lngSlovak, lngSlovenian, lngAlbanian, lngSerbian, lngSwedish,
    lngMalgasy, lngNorwegian, lngOccitan, lngPortuguese, lngPolish, lngRomanian,
    lngRussian, lngSanskrit, lngSlovak, lngSlovenian, lngAlbanian, lngSerbian, lngSwedish,
    lngSyriac, lngTurkish, lngTahitian, lngUkrainian, lngVietnamese, lngChinese, lngDutch,
    lngThai, lngBulgarian, lngBelarusian, lngEstonian, lngLatvian, lngMacedonian,
    lngPashto );

Some basic types and functions need extended RTTI information languages handled by this mORMoti18n unit
- include all languages known by WinXP SP2 without some unicode-only very rare languages; total count is 60
- some languages (Japanase, Chinese, Arabic) may need specific language pack installed on western/latin version of windows
- lngEnglish is the default language of the executable, used as reference for all other translation, and included into executable (no EN.msg file will never be loaded)

Constants implemented in the mORMoti18n unit

LanguageAbr: packed array[TLanguages] of RawByteString =
  ('he', 'gr', 'la', 'ar', 'br', 'ca', 'co', 'cs', 'cp', 'cu', 'cy', 'da', 'de', 'ar',
  'en', 'es', 'fa', 'fi', 'fr', 'ga', 'gd', 'am', 'hr', 'hu', 'hy', 'id', 'ie', 'is',
  'it', 'ja', 'ko', 'bo', 'lt', 'mg', 'no', 'oc', 'pt', 'pl', 'ro', 'ru', 'sa', 'sk',
  'sl', 'sq', 'sn', 'sv', 'sy', 'tr', 'ty', 'uk', 'vi', 'zh', 'nl',
  'th', 'bg', 'be', 'et', 'lv', 'mk', 'ap');

ISO 639-1 compatible abbreviations (not to be translated):

LanguageAlpha: packed array[TLanguages] of byte = (3, 21, 59, 13, 55, 54, 31, 4, 5, 6,
  8, 7, 9, 10, 11, 12, 14, 15, 56, 16, 17, 18, 19, 20, 1, 0, 22, 23, 24, 25, 26, 27, 28,
  29, 30, 2, 32, 57, 33, 58, 52, 34, 35, 37, 36, 38, 39, 40, 41, 42, 43, 44, 45, 46, 53,
  47, 48, 49, 50, 51);

To sort in alphabetic order : LanguageAbr[TLanguages(LanguageAlpha[lng])]
- recreate these table with ModifiedLanguageAbr if LanguageAbr[] changed

LANGUAGE_NONE = TLanguages(255);

Value stored into a TLanguages enumerate to mark no language selected yet

LCID_US = $0409;

US English Windows LCID, i.e. standard international settings

RegistryCompanyName = ''; Language is read from registry once at startup: the sub-entry used to store the i18n settings in the registry; change this value to your company's name, with a trailing backslash ('WorldCompany\' e.g.). the key is HKEY_CURRENT_USER\Software\[RegistryCompanyName]18n\programname

Functions or procedures implemented in the mORMoti18n unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateTime2S</td>
<td>Convert a custom date/time into a VCL-ready string</td>
<td>2328</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>DateTimeToIso</td>
<td>Generic US/English date/time to VCL text conversion</td>
<td>2328</td>
</tr>
<tr>
<td>GetText</td>
<td>Translate the 'Text' term into current language, with no</td>
<td></td>
</tr>
<tr>
<td>i18nAddLanguageCombo</td>
<td>Add combo-box items, for all available languages on disk</td>
<td>2329</td>
</tr>
<tr>
<td>i18nAddLanguageItems</td>
<td>Add strings items, for all available languages on disk</td>
<td>2329</td>
</tr>
<tr>
<td>i18nAddLanguageMenu</td>
<td>Add sub-menu items to the Menu, for all available languages on disk</td>
<td>2329</td>
</tr>
<tr>
<td>i18nLanguageToRegistry</td>
<td>Save the default language to the registry</td>
<td>2329</td>
</tr>
<tr>
<td>i18nRegistryToLanguage</td>
<td>Get the default language from the registry</td>
<td>2329</td>
</tr>
<tr>
<td>Iso2S</td>
<td>Convert a custom date/time into a VCL-ready string</td>
<td>2329</td>
</tr>
<tr>
<td>LanguageAbrToIndex</td>
<td>LanguageAbrToIndex('GR')=1, e.g.</td>
<td>2329</td>
</tr>
<tr>
<td>LanguageToLCID</td>
<td>Convert a i18n language index into a Windows LCID</td>
<td>2329</td>
</tr>
<tr>
<td>LCIDToLanguage</td>
<td>Convert a Windows LCID into a i18n language</td>
<td>2330</td>
</tr>
<tr>
<td>LoadResString</td>
<td>Our hooked procedure for reading a string resource</td>
<td>2330</td>
</tr>
<tr>
<td>S2U</td>
<td>Convert any generic VCL Text into an UTF-8 encoded String</td>
<td>2330</td>
</tr>
<tr>
<td>U2S</td>
<td>Convert an UTF-8 encoded text into a VCL-ready string</td>
<td>2330</td>
</tr>
<tr>
<td>...</td>
<td>Translate the 'English' term into current language</td>
<td>2330</td>
</tr>
</tbody>
</table>

**function** DateTime2S(const DateTime: TDateTime): string;

*Convert a custom date/time into a VCL-ready string*
- this function must be assigned to i18nDateTimeText global var of SynCommons.pas
- wrapper to Language.DateTimeToText(DateTime) method

**function** DateTimeToIso(const DateTime: TDateTime; DateOnly: boolean): string;

*Generic US/English date/time to VCL text conversion*
- not to be used in your programs: it's just here to allow inlining of TLanguageFile.DateTimeToText/DateTimeToText method

**procedure** GetText(var Text: string);

*Translate the 'Text' term into current language, with no || nor $$[$$]*
- LoadResStringTranslate of our customized system.pas points to this procedure
- therefore, direct use of LoadResStringTranslate() is better in apps
- expect "string" type, i.e. UnicodeString for Delphi 2009 and up
procedure i18nAddLanguageCombo(const MsgPath: TFileName; Combo: TComboBox);

Add combo-box items, for all available languages on disk
- uses internally i18nAddLanguageItems() function above
- current language is selected by default
- the OnClick event will launch Language.LanguageClick to change the current language in the registry

function i18nAddLanguageItems(MsgPath: TFileName; List: TStrings): integer;

Add strings items, for all available languages on disk
- it will search in MsgPath for all *.msg available
- if MsgPath is not set, the current executable directory will be used for searching
- new items are added to List: Strings[] will contain a caption text, ready to be displayed, and
  PstrInt(Objects[]) will be the corresponding language ID
- return the current language index in List.Items[]

procedure i18nAddLanguageMenu(const MsgPath: TFileName; Menu: TMenu);[/* remaining procedures and functions */]
function LCIDToLanguage(LCID: integer): TLanguages;
   Convert a Windows LCID into a i18n language

function LoadResString(ResStringRec: PResStringRec): string;
   Our hooked procedure for reading a string resource
   - the default one in System.pas unit is replaced by this one
   - this function add caching and on the fly translation (if LoadResStringTranslate is defined in SynCommons.pas unit)
   - use "string" type, i.e. UnicodeString for Delphi 2009 and up

function S2U(const Text: string): RawUTF8;
   Convert any generic VCL Text into an UTF-8 encoded String
   - same as SynCommons.StringToUTF8()
   Used for DI-2.3.1.3 (page 2550).

function U2S(const Text: RawUTF8): string;
   Convert an UTF-8 encoded text into a VCL-ready string
   - same as SynCommons.UTF8ToString()
   Used for DI-2.3.1.3 (page 2550).

function _ (const English: WinAnsiString): string;
   Translate the 'English' term into current language
   - you should use resourcestring instead of this function
   - call interenaly GetText() procedure, i.e. LoadResStringTranslate()
   Used for DI-2.3.1.3 (page 2550).

Variables implemented in the mORMoti18n unit

CurrentLanguage: TLanguage = ( Index: LANGUAGE_NONE; CharSet: DEFAULT_CHARSET; CodePage: CODEPAGE_US; LCID: LCID_US );
   The global Language used by the User Interface, as updated by the last SetCurrentLanguage() call

i18nCompareStr: TCompareFunction = nil;
   Use this function to compare string with case sensitivity for the UI
   - use current language for comparison
   - can be used for MBCS strings (with such code pages, it will use windows slow but accurate API)

i18nCompareText: TCompareFunction = nil;
   Use this function to compare string with no case sensitivity for the UI
   - use current language for comparison
   - can be used for MBCS strings (with such code pages, it will use windows slow but accurate API)

i18nToLower: TNormTable;
   A table used for fast conversion to lowercase, according to the current language
   - can NOT be used for MBCS strings (with such code pages, you should use windows slow but accurate API)

i18nToLowerByte: TNormTableByte absolute i18nToLower;
   A table used for fast conversion to lowercase, according to the current language
   - can NOT be used for MBCS strings (with such code pages, you should use windows slow but accurate API)
i18nToUpper: TNormTable;
   A table used for fast conversion to uppercase, according to the current language
   - can NOT be used for MBCS strings (with such code pages, you should use windows slow but
     accurate API)

i18nToUpperByte: TNormTableByte absolute i18nToUpper;
   A table used for fast conversion to uppercase, according to the current language
   - can NOT be used for MBCS strings (with such code pages, you should use windows slow but
     accurate API)

isVista: boolean = false;
   True if this program is running on Windows Vista (tm)
   - used to customize on the fly any TTreeView component, to meet Vista and Seven expectations

Language: TLanguageFile = nil;
   Global variable set by SetCurrentLanguage(), used for translation
   - use this object, and its Language property, to retrieve current UI settings

OnTranslateComponent: function(C: TComponent): boolean of object = nil;
   Global event to be assigned for component translation override
   - the method implementing this event must return true if the translation was handled, or false if
     the translation must be done by the framework

SettingsUS: TFormatSettings;
   International settings from US English $0409
   - useful in any software, if you want to save some content with the default english encoding (e.g.
     floating point values with '.')
27.59. mORMotMidasVCL.pas unit

Purpose: Fill a VCL TClientDataset from TSQLTable/TSQLTableJSON data
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the mORMotMidasVCL unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMotVCL</td>
<td>DB VCL dataset using TSQLTable/TSQLTableJSON data access</td>
<td>2426</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

Types implemented in the mORMotMidasVCL unit

TClientDataSetMode = ( cdsNew, cdsAppend, cdsReplace );

How ToClientDataSet/JSONToClientDataSet functions will fill the TClientDataSet instance

Functions or procedures implemented in the mORMotMidasVCL unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSONToClientDataSet</td>
<td>Convert a JSON result into a new VCL TClientDataSet</td>
<td>2333</td>
</tr>
<tr>
<td>JSONToClientDataSet</td>
<td>Convert a JSON result into an existing VCL TClientDataSet</td>
<td>2333</td>
</tr>
<tr>
<td>JSONToClientDataSet</td>
<td>Convert a JSON result into a new VCL TClientDataSet</td>
<td>2333</td>
</tr>
<tr>
<td>ToClientDataSet</td>
<td>Convert a TSQLTable result into an existing VCL TClientDataSet</td>
<td>2333</td>
</tr>
<tr>
<td>ToClientDataSet</td>
<td>For GetDBField() convert a TSQLTable result into a new VCL TClientDataSet</td>
<td>2333</td>
</tr>
</tbody>
</table>

function JSONToClientDataSet(aOwner: TComponent; const aJSON: RawUTF8; const Tables: array of TSQLRecordClass; aClient: TSQLRest=nil; aForceWideString: boolean=false): TClientDataSet; overload;

Convert a JSON result into a new VCL TClientDataSet
- this overloaded method allows to specify the TSQLRecord class types associated with the supplied JSON
function JSONToClientDataSet(aDataSet: TClientDataSet; const aJSON: RawUTF8; aClient: TSQLRest=nil; aMode: TClientDataSetMode=cdsReplace; aLogChange: boolean=false; aForceWideString: boolean=false): boolean; overload;

Convert a JSON result into an existing VCL TClientDataSet
- current implementation will return a TClientDataSet instance, created from the supplied TSQLTable content (a more optimized version may appear later)
- with non-Unicode version of Delphi, you can set aForceWideString to force the use of WideString fields instead of AnsiString, if needed
- with Unicode version of Delphi (2009+), UnicodeString will be used
- for better speed with Delphi older than Delphi 2009 Update 3, it is recommended to use http://andy.jgknet.de/blog/bugfix-units/midas-speed-fix-12

function JSONToClientDataSet(aOwner: TComponent; const aJSON: RawUTF8; aClient: TSQLRest=nil; aForceWideString: boolean=false): TClientDataSet; overload;

Convert a JSON result into a new VCL TClientDataSet
- current implementation will return a TClientDataSet instance, created from the supplied TSQLTable content - see mORMotVCL.pas if you need a more efficient, but read-only version
- with non-Unicode version of Delphi, you can set aForceWideString to force the use of WideString fields instead of AnsiString, if needed
- with Unicode version of Delphi (2009+), UnicodeString will be used
- for better speed with Delphi older than Delphi 2009 Update 3, it is recommended to use http://andy.jgknet.de/blog/bugfix-units/midas-speed-fix-12

function ToClientDataSet(aOwner: TComponent; aTable: TSQLTable; aClient: TSQLRest=nil; aForceWideString: boolean=false): TClientDataSet; overload;

For GetDBField() convert a TSQLTable result into a new VCL TClientDataSet
- current implementation will return a TClientDataSet instance, created from the supplied TSQLTable content (a more optimized version may appear later)
- with non-Unicode version of Delphi, you can set aForceWideString to force the use of WideString fields instead of AnsiString, if needed
- for better speed with Delphi older than Delphi 2009 Update 3, it is recommended to use http://andy.jgknet.de/blog/bugfix-units/midas-speed-fix-12

function ToClientDataSet(aDataSet: TClientDataSet; aTable: TSQLTable; aClient: TSQLRest=nil; aMode: TClientDataSetMode=cdsReplace; aLogChange: boolean=false; aForceWideString: boolean=false): boolean; overload;

Convert a TSQLTable result into an existing VCL TClientDataSet
- current implementation will return a TClientDataSet instance, created from the supplied TSQLTable content (a more optimized version may appear later)
- with non-Unicode version of Delphi, you can set aForceWideString to force the use of WideString fields instead of AnsiString, if needed
- for better speed with Delphi older than Delphi 2009 Update 3, it is recommended to use http://andy.jgknet.de/blog/bugfix-units/midas-speed-fix-12
### 27.60. mORMotMongoDB.pas unit

*Purpose:* Direct optimized MongoDB access for mORMot’s ORM  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

#### Units used in the mORMotMongoDB unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| mORMot          | Common ORM and SOA classes for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1899 |
| SynCommons      | Common functions used by most Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 717  |
| SynLog          | Logging functions used by Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1363 |
| SynMongoDB      | MongoDB document-oriented database direct access classes  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1396 |
| SynTable        | Filter/database/cache/cache/buffer/security/search/multithread/OS features  
- as a complement to SynCommons, which tended to increase too much  
- licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1721 |

#### mORMotMongoDB class hierarchy

- EORMException
- EORMMongoDBException
- TSQlRestStorage
- TSQlRestStorageMongoDB

#### Objects implemented in the mORMotMongoDB unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EORMMongoDBException</td>
<td>For TSynTableStatement exeception class raised by this units</td>
<td>2334</td>
</tr>
<tr>
<td>TSQlRestStorageMongoDB</td>
<td>REST server with direct access to a MongoDB external database</td>
<td>2335</td>
</tr>
</tbody>
</table>

**Note:**

EORMMongoDBException = class(EORMException)  
*For TSynTableStatement exeception class raised by this units*
**TSQLRestStorageMongoDB** = class(TSQLRestStorage)

REST server with direct access to a MongoDB external database
- handle all REST commands via direct SynMongoDB call
- is used by TSQLRestServer.URI for faster RESTful direct access
- JOINed SQL statements are not handled yet

**constructor** Create(aClass: TSQLRecordClass; aServer: TSQLRestServer); **override**

*Initialize the direct access to the MongoDB collection*
- in practice, you should not have to call this constructor, but rather StaticMongoDBRegister() with a TMongoDatabase instance

**destructor** Destroy; **override**

*Release used memory*

**function** CreateSQLMultiIndex(Table: TSQLRecordClass; const FieldNames: array of RawUTF8; Unique: boolean; IndexName: RawUTF8=''): boolean; **override**

*Create one index for all specific FieldNames at once*

**function** EngineDelete(TableModelIndex: integer; ID: TID): boolean; **override**

*Delete a row, calling the current MongoDB server*
- made public since a TSQLRestStorage instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

**function** RetrieveBlobFields(Value: TSQLRecord): boolean; **override**

*Overridden method for one single read call to the MongoDB server*

**function** TableHasRows(Table: TSQLRecordClass): boolean; **override**

*Check if there is some data rows in a specified table*

**function** TableRowCount(Table: TSQLRecordClass): Int64; **override**

*Get the row count of a specified table*
- return -1 on error
- return the row count of the table on success

**function** UpdateBlobFields(Value: TSQLRecord): boolean; **override**

*Overridden method for one single update call to the MongoDB server*

**procedure** Drop;

*Drop the whole table content*
- in practice, dropping the whole MongoDB database would be faster
- but you can still add items to it - whereas Collection.Drop would trigger GPF issues

**procedure** SetEngineAddComputeIdentifier(aIdentifier: word);

*Initialize an internal time-based unique ID generator, linked to a genuine process identifier*
- will allocate a local TSynUniqueIdentifierGenerator
- EngineAddCompute would be set to eaCSynUniqueIdentifier

**property** Collection: TMongoCollection **read** fCollection;

*The associated MongoDB collection instance*
**property** EngineAddCompute: TSQLRestStorageMongoDBEngineAddComputeID read fEngineAddCompute write fEngineAddCompute;

*How the next ID would be compute at each insertion*
- default eacLastIDOnce may be the fastest, but other options are available, and may be used in some special cases
- consider using SetEngineAddComputetIdentifier() which is both safe and fast, with a cloud of servers sharing the same MongoDB collection

**Types implemented in the **mORMotMongoDB** unit**

```pascal
TSQLRestStorageMongoDBEngineAddComputeID = ( eacLastIDOnce, eacLastIDEachTime, eacMaxIDOnce, eacMaxIDEachTime, eacSynUniqueIdentifier );
```

*How TSQLRestStorageMongoDB would compute the next ID to be inserted*
- you may choose to retrieve the last inserted ID via
  ```json
  {$query:{},$orderby:{_id:-1}}
  ```
- or search for the current maximum ID in the collection via
  ```json
  {$group:{_id:null,max:{$max:$_id}}}"
  ```
- eacLastIDOnce and eacMaxIDOnce would execute the request once when the storage instance is first started, whereas eacLastIDEachTime and eacMaxIDEachTime would be execute before each insertion
- with big amount of data, retrieving the maximum ID (eacMaxID*) performs a full scan, which would be very slow: the last inserted ID (eacLastID*) would definitively be faster
- in all cases, to ensure that a centralized MongoDB server has unique ID, you should better pre-compute the ID using your own algorithm depending on your nodes topology, and not rely on the ORM, e.g. using SetEngineAddComputetIdentifier() method, which would allocate a TSynUniqueIdentifierGenerator and associate eacSynUniqueIdentifier

```pascal
TStaticMongoDBRegisterOption = ( mrDoNotRegisterUserGroupTables, mrMapAutoFieldsIntoSmallerLength );
```

*All possible options for StaticMongoDBRegisterAll/TSQLRestMongoDBCreate functions*
- by default, TSQLAuthUser and TSQLAuthGroup tables will be handled via the external DB, but you can avoid it for speed when handling session and security by setting mrDoNotRegisterUserGroupTables
- you can set mrMapAutoFieldsIntoSmallerLength to compute a field name mapping with minimal length, so that the stored BSON would be smaller: by definition, ID/RowID will be mapped as 'id', but other fields will use their first letter, and another other letter if needed (after a '_', or in uppercase, or the next one) e.g. FirstName -> 'f', LastName -> 'l', LockedAccount: 'la'... - WARNING: not yet implemented

```pascal
TStaticMongoDBRegisterOptions = set of TStaticMongoDBRegisterOption;
```

*Set of options for StaticMongoDBRegisterAll/TSQLRestMongoDBCreateOption*

**Functions or procedures implemented in the **mORMotMongoDB** unit**

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>StaticMongoDBRegister</td>
<td>Creates and register a static class on the Server-side to let a given ORM</td>
<td>2337</td>
</tr>
<tr>
<td>class be stored on a remote MongoDB server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StaticMongoDBRegisterAll</td>
<td>Create and register ALL classes of a given model to access a MongoDB server</td>
<td>2337</td>
</tr>
<tr>
<td>Functions or procedures</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TSQLRestMongoDBCreate</td>
<td>Create a new TSQLRest instance, possibly using MongoDB for its ORM process</td>
<td>2338</td>
</tr>
</tbody>
</table>

**Function** StaticMongoDBRegister(aClass: TSQLRecordClass; aServer: TSQLRestServer; aMongoDatabase: TMongoDatabase; aMongoCollectionName: RawUTF8=''; aMapAutoFieldsIntoSmallerLength: boolean=false): TSQLRestStorageMongoDB;

Creates and register a static class on the Server-side to let a given ORM class be stored on a remote MongoDB server
- will associate the supplied class with a MongoDB collection for a specified MongoDB database
- to be called before Server.CreateMissingTables
- by default, the collection name will match TSQLRecord.SQLTableName, but you can customize it with the corresponding parameter
- the TSQLRecord.ID (RowID) field is always mapped to MongoDB's _id field
- will call create needed indexes
- you can later call aServer.InitializeTables to create any missing index and initialize the void tables (e.g. default TSQLAuthGroup and TSQLAuthUser records)
- after registration, you can tune the field-name mapping by calling
  aModel.Props[aClass].ExternalDB.MapField(..)
(just a regular external DB as defined in mORMotDB.pas unit) - it may be a good idea to use short field names on MongoDB side, to reduce the space used for storage (since they will be embedded within the document data)
- it will return the corresponding TSQLRestStorageMongoDB instance - you can access later to it and its associated collection e.g. via:
  (aServer.StaticDataServer[TSQLMyTable] as TSQLRestStorageMongoDB)
- you can set aMapAutoFieldsIntoSmallerLength to compute a field name mapping with minimal length, so that the stored BSON would be smaller: by definition, ID/RowID will be mapped as 'id', but other fields will use their first letter, and another other letter if needed (after a '_', or in uppercase, or the next one) e.g. FirstName -> 'f', LastName -> 'l', LockedAccount: 'la'...

**Function** StaticMongoDBRegisterAll(aServer: TSQLRestServer; aMongoDatabase: TMongoDatabase; aOptions: TStaticMongoDBRegisterOptions=[]; aMongoDBIdentifier: word=0): boolean;

Create and register ALL classes of a given model to access a MongoDB server
- the collection names will follow the class names
- this function will call aServer.InitializeTables to create any missing index or populate default collection content
- if aMongoDBIdentifier is not 0, then SetEngineAddComputeIdentifier() would be called
function TSQLRestMongoDBCreate(aModel: TSQLModel; aDefinition: TSynConnectionDefinition; aHandleAuthentication: boolean; aOptions: TStaticMongoDBRegisterOptions; aMongoDBIdentifier: word=0): TSQLRest; overload;

Create a new TSQLRest instance, possibly using MongoDB for its ORM process.
- If aDefinition.Kind matches a TSQLRest registered class, one new instance of this kind will be created and returned.
- If aDefinition.Kind is 'MongoDB' or 'MongoDBS', it will instantiate an in-memory TSQLRestServerDB or a TSQLRestServerFullMemory instance (calling TSQLRestServer.CreateInMemoryForAllVirtualTables), then call StaticMongoDBRegisterAll() with a TMongoClient initialized from aDefinition.ServerName ('server' or 'server:port') - optionally with TLS enabled if Kind equals 'MongoDBS' - and a TMongoDatabase created from aDefinition.DatabaseName, using authentication if aDefinition.User/Password credentials are set.
- It will return nil if the supplied aDefinition is invalid.
- If aMongoDBIdentifier is not 0, then SetEngineAddComputeIdentifier() would be called for all created TSQLRestStorageMongoDB.
27.61. mORMotMVC.pas unit

*Purpose:* Implements MVC patterns over mORMot's ORM/SOA and SynMustache
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the *mORMotMVC* unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| *mORMot*      | Common ORM and SOA classes for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1899 |
| *mORMotWrappers* | Generate cross-platform clients code and documentation from a mORMot server  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2430 |
| *SynCommons*  | Common functions used by most Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 717  |
| *SynCrypto*   | Fast cryptographic routines (hashing and cypher)  
- implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT  
- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1139 |
| *SynLog*      | Logging functions used by Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1363 |
| *SynMustache* | Logic-less mustache template rendering  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1450 |
### mORMotMVC class hierarchy

#### Objects implemented in the mORMotMVC unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMVCApplication</td>
<td>Exception class triggered by mORMot MVC/MVVM applications externally</td>
<td>2349</td>
</tr>
<tr>
<td>EMVCException</td>
<td>Exception class triggered by mORMot MVC/MVVM applications internally</td>
<td>2349</td>
</tr>
<tr>
<td>TMVCApplication</td>
<td>Defines the main and error pages for the ViewModel of one application</td>
<td>2350</td>
</tr>
<tr>
<td>TMVCAction</td>
<td>Record type to define commands e.g. to redirect to another URI</td>
<td>2346</td>
</tr>
<tr>
<td>IMVCApplication</td>
<td>Defines the main and error pages for the ViewModel of one application</td>
<td>2350</td>
</tr>
<tr>
<td>TMVCRendererAbstract</td>
<td>Abstract MVC rendering execution context</td>
<td>2346</td>
</tr>
<tr>
<td>TMVCRendererFromViews</td>
<td>MVC rendering execution context, returning some rendered View content</td>
<td>2347</td>
</tr>
<tr>
<td>TMVCRendererJson</td>
<td>MVC rendering execution context, returning some un-rendered JSON content</td>
<td>2347</td>
</tr>
<tr>
<td>TMVCRendererReturningData</td>
<td>Abstract MVC rendering execution context, returning some content</td>
<td>2347</td>
</tr>
<tr>
<td>TMVCRun</td>
<td>Abstract class used by TMVCApplication to run</td>
<td>2347</td>
</tr>
<tr>
<td>TMVCRunOnRestServer</td>
<td>Run TMVCApplication directly within a TSQLRestServer method-based service</td>
<td>2348</td>
</tr>
<tr>
<td>TMVCRunWithViews</td>
<td>Abstract class used by TMVCApplication to run TMVCViews-based process</td>
<td>2348</td>
</tr>
<tr>
<td>TMVCSessionAbstract</td>
<td>An abstract class able to implement ViewModel/Controller sessions</td>
<td>2343</td>
</tr>
</tbody>
</table>
Objects | Description | Page
---|---|---
TMVCSessionSingle | Implement a single ViewModel/Controller in-memory session | 2346
TMVCSessionWithCookies | A class able to implement ViewModel/Controller sessions with cookies | 2345
TMVCSessionWithCookieContext | Information used by TMVCSessionWithCookies for cookie generation | 2344
TMVCSessionWithRestServer | Implement a ViewModel/Controller sessions in a TSQLRestServer instance | 2346
TMVCView | Define a particular rendered View | 2341
TMVCViewsAbstract | An abstract class able to implement Views | 2341
TMVCViewsMustache | A class able to implement Views using Mustache templates | 2342
TMVCViewsMustacheParameters | General parameters defining the Mustache Views process | 2342

**TMVCView** = record

*Define a particular rendered View*
- as rendered by TMVCViewsAbstract.Render() method

- Content: RawByteString;
  *The low-level content of this View*

- ContentType: RawUTF8;
  *The MIME content type of this View*

- Flags: TMVCViewFlags;
  *Some additional rendering information about this View*

**TMVCViewsAbstract** = class(TObject)

*An abstract class able to implement Views*

- constructor Create(aInterface: PTypeInfo; aLogClass: TSynLogClass);
  *Initialize the class*

- property Factory: TInterfaceFactory read fFactory;
  *Read-only access to the associated factory for the implementation class*

- property ViewGenerationTimeTag: RawUTF8 read fViewGenerationTimeTag write fViewGenerationTimeTag;
  *Any occurrence of this tag in a rendered view will be converted into the rendering time in microseconds*
  - equals '[[GENERATION_TIME_TAG]]' by default

- property ViewStaticFolder: TFileName read fViewStaticFolder;
  *Retrieve the .static local folder name*
property ViewTemplateFolder: TFileName  read fViewTemplateFolder  
write SetViewTemplateFolder;

Read-only access to the local folder containing the Mustache views

TMVCViewsMustacheParameters = record
  General parameters defining the Mustache Views process
  - used as a separate value so that we would be able to store the settings in a file, e.g. encoded as a JSON object

  CSVExtensions: TFileName;
  The file extensions to search in the given Folder, specified as CSV
  - if not set, will search for 'html, json, css'

  ExtensionForNotExistingTemplate: TFileName;
  File extension (e.g. '.html') to be used to create void templates
  - default '' will create no void template file in the given Folder

  FileTimestampMonitorAfterSeconds: cardinal;
  Defines if the view files should be checked for modification
  - any value would automatically update the rendering template, if the file changed after a given number of seconds - default is 5 seconds
  - setting 0 would be slightly faster, since content would never be checked

  Folder: TFileName;
  Where the mustache template files are stored
  - if not set, will search in a 'Views' folder under the current executable

  Helpers: TSynMustacheHelpers;
  Set of block helpers to be registered to TSynMustache
  - default will use TSynMustache.HelpersGetStandardList definition

TMVCViewsMustache = class(TMVCViewsAbstract)
  A class able to implement Views using Mustache templates

constructor Create(aInterface: PTypeInfo; aLogClass: TSynLogClass=nil; 
aExtensionForNotExistingTemplate: TFileName=''); overload;

  Create an instance of this ViewModel implementation class
  - this overloaded version will use default parameters (i.e. search for html+json+css in the "Views" sub-folder under the executable)
  - will search and parse the matching views (and associated *.partial), optionally creating void templates for any missing view

constructor Create(aInterface: PTypeInfo; const aParameters: 
TMVCViewsMustacheParameters; aLogClass: TSynLogClass=nil); reintroduce; overload;

virtual;

  Create an instance of this ViewModel implementation class
  - define the associated REST instance, the interface definition and the local folder where the mustache template files are stored
  - will search and parse the matching views (and associated *.partial)
**Synopse mORMot Framework**  
Software Architecture Design 1.18  
Date: September 16, 2020

---

**TMVCSessionAbstract** = class(TObject)

An abstract class able to implement ViewModel/Controller sessions  
- see TMVCSessionWithCookies to implement cookie-based sessions  
- this kind of ViewModel will implement client side storage of sessions, storing any (simple) record content on the browser client side  
- at login, a record containing session-related information (session ID, display and login name, preferences, rights...) can be computed only once on the server side from the Model, then stored on the client side (typically in a cookie): later on, session information can be retrieved by the server logic (via CheckAndRetrieve - note that any security attribute should be verified against the Model), then the renderer (CheckAndRetrieveInfo returning the record as TDocVariant in the data context "Session" field) - such a pattern is very efficient and allows good scaling  
- session are expected to be tied to the TMVCSessionAbstract instance lifetime, so are lost after server restart, unless they are persisted via LoadContext/SaveContext methods

---

**destructor** Destroy; **override**;

*Finalize the instance*

**function** RegisterExpressionHelpers(const aNames: array of RawUTF8; const aEvents: array of TSynMustacheHelperEvent): TMVCViewsMustache;

*Define the supplied Expression Helpers definition*
- returns self so that may be called in a fluent interface

**function** RegisterExpressionHelpersForCrypto: TMVCViewsMustache;

*Define some Expression Helpers for hashing*
- i.e. md5, sha1 and sha256 hashing
- would allow e.g. to compute a Gravatar URI via:
  `<img src=http://www.gravatar.com/avatar/{{md5 email}}?s=200 ></img>`
- returns self so that may be called in a fluent interface

**function** RegisterExpressionHelpersForTables(aRest: TSQLRest; const aTables: array of TSQLRecordClass): TMVCViewsMustache; overload;

*Define Expression Helpers for some ORM tables*
- e.g. to read a TSQLMyRecord from its ID value and put its fields in the current rendering data context, you can write:
  aView.RegisterExpressionHelpersForTables(aServer,[TSQLMyRecord]);
- then use the following Mustache tag
  `{{#TSQLMyRecord MyRecordID}} ... {{/TSQLMyRecord MyRecordID}}`
- returns self so that may be called in a fluent interface

**function** RegisterExpressionHelpersForTables( aRest: TSQLRest): TMVCViewsMustache; overload;

*Define Expression Helpers for all ORM tables of the supplied model*
- e.g. to read a TSQLMyRecord from its ID value and put its fields in the current rendering data context, you can write:
  aView.RegisterExpressionHelpersForTables(aServer);
- then use the following Mustache tag
  `{{#TSQLMyRecord MyRecordID}} ... {{/TSQLMyRecord MyRecordID}}`
- returns self so that may be called in a fluent interface
constructor Create; virtual;
Create an instance of this ViewModel implementation class

function CheckAndRetrieve(PRecordData: pointer=nil; PRecordTypeInfo: pointer=nil; PExpires: PCardinal=nil): integer; virtual; abstract;
Retrieve the current session ID
- can optionally retrieve the associated record Data parameter

function CheckAndRetrieveInfo(PRecordDataTypeInfo: pointer): variant; virtual;
Retrieve the session information as a JSON object
- returned as a TDocVariant, including any associated record Data
- will call CheckAndRetrieve() then RecordSaveJSON() and _JsonFast()

function Exists: boolean; virtual; abstract;
Fast check if there is a session associated to the current context

function Initialize(PRecordData: pointer=nil; PRecordTypeInfo: pointer=nil; SessionTimeOutMinutes: cardinal=60): integer; virtual; abstract;
Will create a new session
- setting an optional record data, and returning the internal session ID
- you can supply a time period, after which the session will expire - default is 1 hour - note that overridden methods may not implement it

function LoadContext(const Saved: RawUTF8): boolean; virtual; abstract;
Restore session generation information from SaveContext format
- returns TRUE on success

function SaveContext: RawUTF8; virtual; abstract;
Return all session generation information as ready-to-be stored string
- to be retrieved via LoadContext, e.g. after restart

procedure Finalize; virtual; abstract;
Clear the session

TMVCSessionWithCookiesContext = packed record
Information used by TMVCSessionWithCookies for cookie generation
- i.e. the session ID, cookie name, encryption and HMAC secret keys
- this data can be persisted so that the very same cookie information are available after server restart

CookieName: RawUTF8;
The cookie name, used for storage on the client side

Crypt: array[byte] of byte;
Secret information, used for encryption of the cookie content

CryptNonce: Cardinal;
Random IV used as CTR on Crypt[] secret key

Secret: THMAC_CRC32C;
Secret information, used for HMAC digital signature of cookie content
SessionSequence: integer;
    An increasing counter, to implement unique session ID

**TMVCSessionWithCookies** = class(TMVCSessionAbstract)

* A class able to implement ViewModel/Controller sessions with cookies
  * this kind of ViewModel will implement cookie-based sessions, able to store any (simple) record content in the cookie, on the browser client side
  * those cookies have the same feature set than JWT, but with a lower payload (thanks to binary serialization), and cookie safety (not accessible from JavaScript): they are digitally signed (with HMAC-CRC32C and a temporary secret key), they include an unique session identifier (like "jti" claim), issue and expiration dates (like "iat" and "exp" claims), and they are encrypted with a temporary key - this secret keys is tied to the TMVCSessionWithCookies instance lifetime, so new cookies are generated after server restart, unless they are persisted via LoadContext/SaveContext
  * signature and encryption are weak, but very fast, to avoid DDOS attacks

**constructor** Create; override;
    Create an instance of this ViewModel implementation class

**function** CheckAndRetrieve(PRecordData: pointer=nil; PRecordTypeInfo: pointer=nil; PExpires: PCardinal=nil): integer; override;
    * Retrieve the session ID from the current cookie
      * can optionally retrieve the record Data parameter stored in the cookie
      * will return the 32-bit internal session ID, or 0 if the cookie is invalid

**function** Exists: boolean; override;
    * Fast check if there is a cookie session associated to the current context

**function** Initialize(PRecordData: pointer=nil; PRecordTypeInfo: pointer=nil; SessionTimeOutMinutes: cardinal=60): integer; override;
    * Will initialize the session cookie
      * setting an optional record data, which will be stored Base64-encoded
      * will return the 32-bit internal session ID
      * you can supply a time period, after which the session will expire - default is 1 hour, and could go up to

**function** LoadContext(const Saved: RawUTF8): boolean; override;
    * Restore cookie generation information from SaveContext text format
      * returns TRUE after checking the crc and unserializing the supplied data
      * WARNING: if the unerlying record type structure changed (i.e. any field is modified or added), restoration will lead to data corruption of low-level binary content, then trigger unexpected GPF: if you change the record type definition, do NOT use LoadContext - and reset all cookies

**function** SaveContext: RawUTF8; override;
    * Return all cookie generation information as base64 encoded text
      * to be retrieved via LoadContext

**procedure** Finalize; override;
    * Clear the session
      * by deleting the cookie on the client side

---

mORMotMVC.pas unit - Rev. 1.18  
Page 2346 of 2552
property Context: TMVCSessionWithCookiesContext read fContext write fContext;
  Direct access to the low-level information used for cookies generation
  - use SaveContext and LoadContext methods to persist this information before server shutdown,
  so that the cookies can be re-used after restart

property CookieName: RawUTF8 read fContext.CookieName write fContext.CookieName;
  You can customize the cookie name
  - default is 'mORMot', and cookie is restricted to Path=/RestRoot

TMVCSessionWithRestServer = class(TMVCSessionWithCookies)
  Implement a ViewModel/Controller sessions in a TSQLRestServer instance
  - will use ServiceContext.Request threadvar to access the client cookies

TMVCSessionSingle = class(TMVCSessionWithCookies)
  Implement a single ViewModel/Controller in-memory session
  - this kind of session could be used in-process, e.g. for a VCL/FMX GUI
  - do NOT use it with multiple clients, e.g. from HTTP remote access

TMVCAction = record
  Record type to define commands e.g. to redirect to another URI
  - do NOT access those record property directly, but rather use
    TMVCApplication.GotoView/GotoError/GotoDefault methods, e.g.
    function TBlogApplication.Logout: TMVCAction;
    begin
    CurrentSession.Finalize;
    GotoDefault(result);
    end;
  - this record type should match exactly TServiceCustomAnswer layout, so that
    TServiceMethod.InternalExecute() would handle it directly

  RedirectToMethodName: RawUTF8;
    The method name to be executed

  RedirectToMethodParameters: RawUTF8;
    May contain a JSON object which will be used to specify parameters to the specified method

  ReturnedStatus: cardinal;
    Which HTTP Status code should be returned
    - if RedirectToMethodName is set, will return 307 HTTP_TEMPORARYREDIRECT by default, but you
      can set here the expected HTTP Status code, e.g. 201 HTTP_CREATED or 404 HTTP_NOTFOUND

TMVCRendererAbstract = class(TObject)
  Abstract MVC rendering execution context
  - you should not execute this abstract class, but any of the inherited class
  - one instance inherited from this class would be allocated for each event
  - may return some data (when inheriting from TMVCRendererReturningData), or even simply
    display the value in a VCL/FMX GUI, without any output

  constructor Create(aApplication: TMVCApplication); reintroduce;
    Initialize a rendering process for a given MVC Application/ViewModel
procedure ExecuteCommand(aMethodIndex: integer); virtual;
   Main execution method of the rendering process
   - Input should have been set with the incoming execution context

property Input: RawUTF8 read fInput write fInput;
   Incoming execution context, to be processed via ExecuteCommand() method
   - should be specified as a raw JSON object

TMVCRendererReturningData = class(TMVCRendererAbstract)
   Abstract MVC rendering execution context, returning some content
   - the Output property would contain the content to be returned
   - can be used to return e.g. some rendered HTML or some raw JSON, or even some server-side
generated report as PDF, using our mORMotReport.pas
   constructor Create(aRun: TMVCRunWithViews); reintroduce; virtual;
   Initialize a rendering process for a given MVC Application/ViewModel
   - you need to specify a MVC Views engine, e.g. TMVCViewsMustache instance
   procedure ExecuteCommand(aMethodIndex: integer); override;
   Main execution method of the rendering process
   - this overridden method would handle proper caching as defined by
   TMVCRunWithViews.SetCache()
   property Output: TServiceCustomAnswer read fOutput;
   Caller should retrieve this value after ExecuteCommand method execution

TMVCRendererFromViews = class(TMVCRendererReturningData)
   MVC rendering execution context, returning some rendered View content
   - will use an associated Views templates system, e.g. a Mustache renderer
   constructor Create(aRun: TMVCRunWithViews); override;
   Initialize a rendering process for a given MVC Application/ViewModel
   - this overridden constructor will ensure that cache is enabled
TMVCRendererJson = class(TMVCRendererReturningData)
   MVC rendering execution context, returning some un-rendered JSON content
   - may be used e.g. for debugging purpose
   - for instance, TMVCRunOnRestServer will return such context with the supplied URI ends with
   '/json' (e.g. for any /root/method/json request)

TMVCRun = class(TObject)
   Abstract class used by TMVCApplication to run
   - a single TMVCApplication logic may handle several TMVCRun instances
   constructor Create(aApplication: TMVCApplication); reintroduce;
   Link this runner class to a specified MVC application
   - will also reset the associated Application.Session instance
procedure NotifyContentChanged; virtual;
Method called to flush the caching mechanism for all MVC commands

procedure NotifyContentChangedForMethod(const aMethodName: RawUTF8); overload;
You may call this method to flush any caching mechanism for a MVC command

procedure NotifyContentChangedForMethod(aMethodIndex: integer); overload; virtual;
You may call this method to flush any caching mechanism for a MVC command

property Application: TMVCApplication read fApplication write fApplication;
Read-write access to the associated MVC Application/ViewModel instance

TMVCRunWithViews = class(TMVCRun)
Abstract class used by TMVCApplication to run TMVCViews-based process
- this inherited class will host a MVC Views instance, and handle an optional simple in-memory cache

constructor Create(aApplication: TMVCApplication; aViews: TMVCViewsAbstract=nil);
reintroduce;
Link this runner class to a specified MVC application

destructor Destroy; override;
Finalize this instance

function SetCache(const aMethodName: RawUTF8; aPolicy: TMVCRendererCachePolicy;
aTimeOutSeconds: cardinal=0): TMVCRunWithViews; virtual;
Defines the caching policy for a given MVC command
- a time expiration period (up to 5 minutes) can also be defined per MVC command - leaving default 0 will set to 5 minutes expiration delay
- function calls can be chained to create some fluent definition interface like in
TAnyBLogapplication.Create:
  fMainRunner := TMVCRunWithViews.Create(self).SetCache('default',cacheRoot);

procedure NotifyContentChangedForMethod(aMethodIndex: integer); override;
Method called to flush the caching mechanism for a MVC command

property Views: TMVCViewsAbstract read fViews;
Read-write access to the associated MVC Views instance

TMVCRunOnRestServer = class(TMVCRunWithViews)
Run TMVCApplication directly within a TSQLRestServer method-based service
- this is the easiest way to host and publish a MVC Application, optionally in conjunction with
REST/AJAX client access
constructor Create(aApplication: TMVCAppllication; aRestServer: TSQLRestServer=nil; const aSubURI: RawUTF8=''; aViews: TMVCViewsAbstract=nil; aPublishOptions: TMVCPublishOptions= [low(TMVCPublishOption)..high(TMVCPublishOption)]);
            reintroduce;
            This constructor will publish some views to a TSQLRestServer instance
            - the associated RestModel can match the supplied TSQLRestServer, or be another instance (if the data model is not part of the publishing server)
            - all TMVCAppllication methods would be registered to the TSQLRestServer, as /root/methodName if aSubURI is '', or as /root/aSubURI/methodName
            - if aApplication has no Views instance associated, this constructor will initialize a Mustache renderer in its default folder, with '.html' void template generation
            - will also create a TMVCSessionWithRestServer for simple cookie sessions
            - aPublishOptions could be used to specify integration with the server

function AddStaticCache(const aFileName: TFileName; const aFileContent: RawByteString): RawByteString;
            Define some content for a static file
            - only used if cacheStatic has been defined

property PublishOptions: TMVCPublishOptions read fPublishOptions write fPublishOptions;
            Current publishing options, as specify to the constructor

property StaticCacheControlMaxAge: integer read fStaticCacheControlMaxAge write fStaticCacheControlMaxAge;
            Optional "Cache-Control: max-age=###" header value for static content

EMVCException = class(ESynException)
            Exception class triggered by mORMot MVC/MVVM applications internally
            - those error are internal fatal errors of the server side process

EMVCApplication = class(ESynException)
            Exception class triggered by mORMot MVC/MVVM applications externally
            - those error are external errors which should be notified to the client
            - can be used to change the default view, e.g. on application error

constructor CreateDefault(aStatus: cardinal=HTTP_TEMPORARYREDIRECT);
            Same as calling TMVCAppllication.GotoDefault
            - HTTP_TEMPORARYREDIRECT will change the URI, but HTTP_SUCCESS won't

constructor CreateGotoError(aHtmlErrorCode: integer); overload;
            Same as calling TMVCAppllication.GotoError()

constructor CreateGotoError(const aErrorMessage: string; aErrorCode: integer=HTTP_BADREQUEST); overload;
            Same as calling TMVCAppllication.GotoError()

constructor CreateGotoView(const aMethod: RawUTF8; const aParametersNameValuePairs: array of const; aStatus: cardinal=HTTP_TEMPORARYREDIRECT);
            Same as calling TMVCAppllication.GotoView()
            - HTTP_TEMPORARYREDIRECT will change the URI, but HTTP_SUCCESS won't
IMVCApplication = interface(IInvokable)
  Defines the main and error pages for the ViewModel of one application
  
procedure Default(var Scope: variant);
  The default main page
  - whole data context is retrieved and returned as a TDocVariant

procedure Error(var Msg: RawUTF8; var Scope: variant);
  The error page
  - in addition to the error message, a whole data context is retrieved and returned as a TDocVariant

TMVCApplication = class(TInjectableObject)
  Parent class to implement a MVC/MVVM application
  - you should inherit from this class, then implement an interface inheriting from IMVCApplication to define the various commands of the application
  - here the Model would be a TSQLRest instance, Views will be defined by TMVCViewsAbstract (e.g. TMVCViewsMustache), and the ViewModel/Controller will be implemented with IMVCApplication methods of the inherited class
  - inherits from TInjectableObject, so that you could resolve dependencies via services or stubs, following the IoC pattern
  
  destructor Destroy; override;
  Finalize the application
  - and release any associated CurrentSession, Views, and fMainRunner

procedure Start(aRestModel: TSQLRest; aInterface: PTypeInfo); virtual;
  Initialize the instance of the MVC/MVVM application
  - define the associated REST instance, and the interface definition for application commands
  - is not defined as constructor, since this TInjectableObject may expect injection using the CreateInjected() constructor

property CurrentSession: TMVCSessionAbstract read fSession write SetSession;
  Read-write access to the associated Session instance

property Factory: TInterfaceFactory read fFactory;
  Read-only access to the associated factory for IMVCApplication interface

property Locker: IAutoLocker read fLocker;
  Global mutex which may be used to protect ViewModel/Controller code
  - you may call Locker.ProtectMethod in any implementation method to ensure that no other thread would access the same data
  - for store some cache data among methods, you may consider defining a ILockedDocVariant private field, and use it to store values safely
  - note that regular RestModel CRUD operations are already thread safe, so it is not necessary to use this Locker with ORM or SOA methods
property MainRunner: TMVCRun read fMainRunner;

Read-write access to the main associated TMVCRun instance
- if any TMVCRun instance is stored here, will be freed by Destroy
- but note that a single TMVCApplication logic may handle several TMVCRun

property RestModel: TSQLRest read fRestModel;

Read-only access to the associated mORMot REST instance implementing the MVC data Model of the application
- is a TSQLRestServer instance e.g. for TMVCRunOnRestServer

Types implemented in the mORMotMVC unit

TMVCPublishOption = ( publishMvcInfo, publishStatic, cacheStatic, registerORMTableAsExpressions, bypassAuthentication );

The kinds of optional content which may be published
- publishMvcInfo will define a /root/[aSubURI/]mvc-info HTML page, which is pretty convenient when working with views
- publishStatic will define a /root/[aSubURI/].static sub-folder, ready to serve any file available in the Views/.static local folder, via an in-memory cache (if cacheStatic is also defined)
- cacheStatic enables an in-memory cache of publishStatic files; if not set, TSQLRestServerURIContext.ReturnFile is called to avoid buffering, which may be a better solution on http.sys or if NGINX's X-Accel-Redirect header is set
- registerORMTableAsExpressions will register Mustache Expression Helpers for every TSQLRecord table of the Server data model
- by default, TSQLRestServer authentication would be by-passed for all MVC routes, unless bypassAuthentication option is undefined

TMVCPublishOptions = set of TMVCPublishOption;

Which kind of optional content should be publish

TMVCRendererCachePolicy = ( cacheNone, cacheRootIgnoringSession, cacheRootIfSession, cacheRootIfNoSession, cacheRootWithSession, cacheWithParametersIgnoringSession, cacheWithParametersIfSession, cacheWithParametersIfNoSession );

How TMVCRendererReturningData should cache its content

TMVCViewFlags = set of (viewHasGenerationTimeTag);

TMVCView.Flags rendering context

Constants implemented in the mORMotMVC unit

MVCINFO_URI = 'mvc-info';

The pseudo-method name for the MVC information html page

STATIC_URI = '.static';

The pseudo-method name for any static content for Views
27.62. mORMotReport.pas unit

*Purpose*: Reporting unit
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The *mORMotReport* unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.3.2</td>
<td>A reporting feature, with full preview and export as PDF or TXT files, shall be integrated</td>
<td>2550</td>
</tr>
</tbody>
</table>

**Units used in the *mORMotReport* unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| SynCommons    | Common functions used by most Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                   | 717  |
| SynGdiPlus    | GDI+ library API access  
- adds GIF, TIF, PNG and JPG pictures read/write support as standard  
- make available most useful GDI+ drawing methods  
- allows Antialiased rending of any EMF file using GDI+  
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                   | 1350 |
| SynLZ         | SynLZ Compression routines  
- licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                                                                                  | 1393 |
| SynPdf        | PDF file generation  
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                       | 1473 |

**Objects implemented in the *mORMotReport* unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TColRec</td>
<td>Internal format of a text column</td>
<td>2353</td>
</tr>
</tbody>
</table>
Objects | Description | Page
--- | --- | ---
TGDIPageContent | Contains one page | 2354
TGDIPagereference | Internal structure used to store bookmarks or links | 2353
TGDIPages | Report class for generating documents from code | 2354
THeaderFooter | Internal format of the header or footer text | 2353
TPagePaintBox | Hack the TPaintBox to allow custom background erase | 2353
TSavedState | A report layout state, as used by SaveLayout/RestoreSavedLayout methods | 2353

**TSavedState = record**

* A report layout state, as used by SaveLayout/RestoreSavedLayout methods

**THeaderFooter = class(TObject)**

* Internal format of the header or footer text

**constructor** Create(Report: TGDIPages; doubleline: boolean; const aText: SynUnicode=''; IsText: boolean=false);

* Initialize the header or footer parameters with current report state

**TColRec = record**

* Internal format of a text column

**TPagePaintBox = class(TPaintBox)**

* Hack the TPaintBox to allow custom background erase

**TGDIPagereference = class(TObject)**

* Internal structure used to store bookmarks or links

**Page**: Integer;

* The associated page number (starting at 1)

**Preview**: TRect;

* Coordinates on screen of the hot zone

**Rect**: TRect;

* Graphical coordinates of the hot zone
  - for bookmarks, Top is the Y position
  - for links, the TRect will describe the hot region
  - for Outline, Top is the Y position and Bottom the outline tree level

**constructor** Create(PageNumber: integer; Left, Top, Right, Bottom: integer);

* Initialize the structure with the current page

**procedure** ToPreview(Pages: TGDIPages);

* Compute the coordinates on screen into Preview
TGDIPageContent = record
  Contains one page
  MarginPx: TRect;
    Margin of the page
  MetaFileCompressed: RawByteString;
    SynLZ-compressed content of the page
  OffsetPx: TPoint;
    Non printable offset of the page
  SizePx: TPoint;
    The physical page size
  Text: string;
    Text equivalent of the page

TGDIPages = class(TScrollBox)
  Report class for generating documents from code
  - data is drawn in memory, they displayed or printed as desired
  - allow preview and printing, and direct pdf export
  - handle bookmark, outlines and links inside the document
  - page coordinates are in mm's

  Used for DI-2.3.2 (page 2550).
  Caption: string;
    The title of the report
    - used for the preview caption form
    - used for the printing document name

  ForceCopyTextAsWholeContent: boolean;
    If true, the headers are copied only once to the text

  ForceInternalAntiAliased: boolean;
    If true, drawing will NOT to use native GDI+ 1.1 conversion
    - we found out that GDI+ 1.1 was not as good as our internal conversion function written in
      Delphi, e.g. for underlined fonts
    - so this property is set to true by default for proper display on screen
    - will only be used if ForceNoAntiAliased is false, of course

  ForceInternalAntiAliasedFontFallBack: boolean;
    If true, internal text drawing will use a font-fallback mechanism for characters not existing within
    the current font (just as with GDI)
    - is disabled by default, but could be set to TRUE to force enabling
      TGDIPlusFull.UseDrawString property

  ForceNoAntiAliased: boolean;
    If true the preview will not use GDI+ library to draw anti-aliased graphics
    - this may be slow on old computers, so caller can disable it on demand
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForcePrintAsBitmap</td>
<td>Boolean; If true, the PrintPages() method will use a temporary bitmap for printing. Some printer device drivers have problems with printing metafiles which contains other metafiles; should have been fixed. Not useful, since slows the printing a lot and makes huge memory usage.</td>
</tr>
<tr>
<td>GroupsMustBeOnSamePage</td>
<td>Boolean; Set group page fill method. If set to true, the groups will be forced to be placed on the same page (this was the original default &quot;Pages&quot; component behavior, but this is not usual in page composition, so is disabled by default in TGDIPages). If set to false, the groups will force a page feed if there is not enough place for 20 lines on the current page (default behavior).</td>
</tr>
<tr>
<td>OnPopupMenuClick</td>
<td>TNotifyEvent; Event triggered when a ReportPopupMenu item is selected. Default handling (i.e. leave this field nil) is for Page navigation. You can override this method for handling additional items to the menu. The Tag component of the custom TMenuItem should be 0 or greater than Report pages count: use 1000 as a start for custom TMenuItem.Tag values.</td>
</tr>
<tr>
<td>OnPopupMenuPopup</td>
<td>TNotifyEvent; Event triggered when the ReportPopupMenu is displayed. Default handling (i.e. leave this field nil) is to add Page navigation. You can override this method for adding items to the ReportPopupMenu.</td>
</tr>
<tr>
<td>OnStringToUnicode</td>
<td>TOnStringToUnicodeEvent; Customize text conversion before drawing. Text content can be modified by this event handler to customize some characters (e.g. ‘&gt;=' can be converted to its Unicode glyph).</td>
</tr>
<tr>
<td>PopupMenuClass</td>
<td>TPopupMenuClass; User can customize this class to create an advanced popup menu instance.</td>
</tr>
<tr>
<td>PreviewSurfaceBitmap</td>
<td>TBitmap; The bitmap used to draw the page.</td>
</tr>
<tr>
<td>constructor</td>
<td>Create(AOwner: TComponent); override; Creates the reporting component.</td>
</tr>
<tr>
<td>destructor</td>
<td>Destroy; override; Finalize the component, releasing all used memory.</td>
</tr>
<tr>
<td>function</td>
<td>AddBookMark(const aBookmarkName: string; aYPosition: integer=0): Boolean; virtual; Create a bookmark entry at the current position of the current page. Return false if this bookmark name was already existing, true on success. If aYPosition is not 0, the current Y position will be used.</td>
</tr>
<tr>
<td>function</td>
<td>CreatePictureMetaFile(Width, Height: integer; out MetaCanvas: TCanvas): TMetaFile; Create a meta file and its associated canvas for displaying a picture. You must release manually both Objects after usage.</td>
</tr>
</tbody>
</table>
function CurrentGroupPosStart: integer;
   Distance (in mm's) from the top of the page to the top of the current group
   - returns Current yp if no group is in use

function ExportPDF(aPdfFileName: TFileName; ShowErrorOnScreen: boolean;
   LaunchAfter: boolean=true): boolean;
   Export the current report as PDF file
   - uses internal PDF code, from Synopse PDF engine (handle bookmarks, outline and twin
     bitmaps) - in this case, a file name can be set

function ExportPDFStream(aDest: TStream): boolean;
   Export the current report as PDF in a specified stream
   - uses internal PDF code, from Synopse PDF engine (handle bookmarks, outline and twin
     bitmaps) - in this case, a file name can be set

function GetColumnInfo(index: integer): TColRec;
   Retrieve the attributes of a specified column

function GotoBookmark(const aBookmarkName: string): Boolean; virtual;
   Go to the specified bookmark
   - returns true if the bookmark name was existing and reached

function HasSpaceFor(mm: integer): boolean;
   Returns true if there is enough space in the current Report for a vertical size, specified in mm

function HasSpaceForLines(Count: integer): boolean;
   Returns true if there is enough space in the current Report for Count lines
   - Used to check if there's sufficient vertical space remaining on the page for the specified
     number of lines based on the current Y position

function MmToPrinter(const R: TRect): TRect;
   Convert a rect of mm into pixel canvas units

function MmToPrinterPxX(mm: integer): integer;
   Convert a mm X position into pixel canvas units

function MmToPrinterPxY(mm: integer): integer;
   Convert a mm Y position into pixel canvas units

function MmToPrinter(const R: TRect): TRect;
   Convert a rect of mm into pixel canvas units

function MmToPrinterPxX(mm: integer): integer;
   Convert a mm X position into pixel canvas units

function MmToPrinterPxY(mm: integer): integer;
   Convert a mm Y position into pixel canvas units

function NewPopupMenuItem(const aCaption: string; Tag: integer=0; SubMenu:
   TMenuItem=nil; OnClick: TNotifyEvent=nil; ImageIndex: integer=-1): TMenuIte
   Add an item to the popup menu
   - used mostly internally to add page browsing
   - default OnClick event is to go to page set by the Tag property

function PrinterPxToMmX(px: integer): integer;
   Convert a pixel canvas X position into mm

function PrinterPxToMmY(px: integer): integer;
   Convert a pixel canvas Y position into mm

function PrinterToMM(const R: TRect): TRect;
   Convert a rect of pixel canvas units into mm
function PrintPages(PrintFrom, PrintTo: integer): boolean;
    Print the selected pages to the default printer of Printer unit
    - if PrintFrom=0 and PrintTo=0, then all pages are printed
    - if PrintFrom=-1 or PrintTo=-1, then a printer dialog is displayed

function TextWidth(const Text: SynUnicode): integer;
    Return the width of the specified text, in mm

function TitleFlags: integer;
    Get the formatting flags associated to a Title

procedure AddColumn(left, right: integer; align: TColAlign; bold: boolean);
    Register a column, with proper alignment

procedure AddColumnHeaders(const headers: array of SynUnicode; WithBottomGrayLine: boolean=false; BoldFont: boolean=false; RowLineHeight: integer=0; flags: integer=0);
    Register some column headers, with the current font formatting
    - Column headers will appear just above the first text output in columns on each page
    - you can call this method several times in order to have diverse font formats across the column headers

procedure AddColumnHeadersFromCSV(var CSV: PWideChar; WithBottomGrayLine: boolean; BoldFont: boolean=false; RowLineHeight: integer=0);
    Register some column headers, with the current font formatting
    - Column headers will appear just above the first text output in columns on each page
    - call this method once with all columns text as CSV

procedure AddColumns(const PercentWidth: array of integer; align: TColAlign=caLeft);
    Register same alignment columns, with percentage of page column width
    - sum of all percent width should be 100, but can be of any value
    - negative widths are converted into absolute values, but corresponding alignment is set to right
    - if a column need to be right aligned or currency aligned, use SetColumnAlign() method below
    - individual column may be printed in bold with SetColumnBold() method

procedure AddLineToFooter(doubleline: boolean);
    Adds either a single line or a double line (drawn between the left & right page margins) to the page footer

procedure AddLineToHeader(doubleline: boolean);
    Adds either a single line or a double line (drawn between the left & right page margins) to the page header

procedure AddLink(const aBookmarkName: string; aRect: TRect; aPageNumber: integer=0; aNoBorder: boolean=false); virtual;
    Create a link entry at the specified coordinates of the current page
    - coordinates are specified in mm
    - the bookmark name is not checked by this method: a bookmark can be linked before being marked in the document
procedure AddOutline(const aTitle: string; aLevel: Integer; aYPosition: integer=0; aPageNumber: integer=0); virtual;

Create an outline entry at the current position of the current page
- if aYPosition is not 0, the current Y position will be used

procedure AddPagesToFooterAt(const PageText: string; XPos: integer; YPosMultiplier: integer=1);

Will add the current 'Page n/n' text at the specified position
- PageText must be of format 'Page %d/%d', in the desired language
- if XPos=-1, will put the text at the current right margin
- if the vertical position does not fit your need, you could set YPosMultiplier to a value which will be multiplied by fFooterHeight to compute the YPos

procedure AddTextToFooter(const s: SynUnicode);

Adds text using to current font and alignment to the page footer

procedure AddTextToFooterAt(const s: SynUnicode; XPos: integer);

Adds text to the page footer at the specified horizontal position and using to current font. No Line feed will be triggered.
- if XPos=-1, will put the text at the current right margin

procedure AddTextToHeader(const s: SynUnicode);

Adds text using to current font and alignment to the page header

procedure AddTextToHeaderAt(const s: SynUnicode; XPos: integer);

Adds text to the page header at the specified horizontal position and using to current font.
- No Line feed will be triggered: this method doesn't increment the YPos, so can be used to add multiple text on the same line
- if XPos=-1, will put the text at the current right margin

procedure AppendRichEdit(RichEditHandle: HWnd; EndOfPagePositions: PIntegerDynArray=nil);

Append a Rich Edit content to the current report
- note that if you want the TRichEdit component to handle more than 64 KB of RTF content, you have to set its MaxLength property as expected (this is a limitation of the VCL, not of this method)
- you can specify optionally a pointer to a TIntegerDynArray variable, which will be filled with the position of each page last char: it may be handy e.g. to add some cross-reference table about the rendered content

procedure BeginDoc;

Begin a Report document
- Every report must start with BeginDoc and end with EndDoc
- note that Printers.SetPrinter() should be set BEFORE calling BeginDoc, otherwise you may have a "canvas does not allow drawing" error

procedure BeginGroup;

Begin a Group: stops the contents from being split across pages
- BeginGroup-EndGroup text blocks can't be nested

procedure Clear; virtual;

Clear the current Report document
procedure ClearColumnHeaders;
    Clear the Headers associated to the Columns

procedure ClearColumns;
    Erase all columns and the associated headers

procedure ClearFooters;
    Clear all already predefined Footers

procedure ClearHeaders;
    Clear all already predefined Headers

procedure ColumnHeadersNeeded;
    ColumnHeadersNeeded will force column headers to be drawn again just prior to printing the next row of columned text
    - Usually column headers are drawn once per page just above the first column.
    - ColumnHeadersNeeded is useful where columns of text have been separated by a number of lines of non-columned text

procedure DrawAngledTextAt(const s: SynUnicode; XPos, Angle: integer);
    Draw one line of text, with a specified Angle and X Position

procedure DrawArrow(Point1, Point2: TPoint; HeadSize: integer; SolidHead: boolean);
    Draw an Arrow

procedure DrawBMP(rec: TRect; bmp: TBitmap); overload;
    Stretch draws a bitmap image at the specified page coordinates in mm's

procedure DrawBMP(bmp: TBitmap; bLeft, bWidth: integer; const Legend: string=''); overload;
    Add the bitmap at the specified X position
    - if there is not enough place to draw the bitmap, go to next page
    - then the current Y position is updated
    - bLeft (in mm) is calculated in reference to the LeftMargin position
    - if bLeft is maxInt, the bitmap is centered to the page width
    - bitmap is stretched (keeping aspect ratio) for the resulting width to match the bWidth parameter (in mm)

procedure DrawBox(left, top, right, bottom: integer);
    Draw a square box at the given coordinates

procedure DrawBoxFilled(left, top, right, bottom: integer; Color: TColor);
    Draw a filled square box at the given coordinates

procedure DrawColumnLine(ColIndex: integer; aAtTop: boolean; aDoDoubleLine: boolean);
    Draw a Line, following a column layout

procedure DrawDashedLine;
    Draw a Dashed Line between the left & right margins
procedure DrawGraphic(graph: TGraphic; bLeft, bWidth: integer; const Legend: SynUnicode='');

Add the graphic (bitmap or metafile) at the specified X position
- handle only TBitmap and TMetafile kind of TGraphic
- if there is not enough place to draw the bitmap, go to next page
- then the current Y position is updated
- bLeft (in mm) is calculated in reference to the LeftMargin position
- if bLeft is maxInt, the bitmap is centered to the page width
- bitmap is stretched (keeping aspect ratio) for the resulting width to match the bWidth parameter (in mm)

procedure DrawLine(doubleline: boolean=false);

Draw a Line, either simple or double, between the left & right margins

procedure DrawLinesInCurrencyCols(doublelines: boolean);

Draw (double if specified) lines at the bottom of all currency columns

procedure DrawMeta(rec: TRect; meta: TMetafile);

Stretch draws a metafile image at the specified page coordinates in mm’s

procedure DrawText(const s: string; withNewLine: boolean=true);

Draw some text as a paragraph, with the current alignment
- this method does all word-wrapping and formatting if necessary
- this method handle multiple paragraphs inside s (separated by newlines - i.e. #13)
- by default, will write a paragraph, unless withNewLine is set to FALSE, so that the next
  DrawText() will continue drawing at the current position

procedure DrawTextAcrossCols(const StringArray, LinkArray: array of SynUnicode; BackgroundColor: TColor=clNone); overload;

Draw some text, split across every columns
- you can specify an optional bookmark name to be used to link a column content via a AddLink() call
- if BackgroundColor is not clNone (i.e. clRed or clNavy or clBlack), the row is printed on white
  with this background color (e.g. to highlight errors)

procedure DrawTextAcrossCols(const StringArray: array of SynUnicode; BackgroundColor: TColor=clNone); overload;

Draw some text, split across every columns
- if BackgroundColor is not clNone (i.e. clRed or clNavy or clBlack), the row is printed on white
  with this background color (e.g. to highlight errors)

procedure DrawTextAcrossColsFromCSV(var CSV: PWideChar; BackgroundColor: TColor=clNone);

Draw some text, split across every columns
- this method expect the text to be separated by commas
- if BackgroundColor is not clNone (i.e. clRed or clNavy or clBlack), the row is printed on white
  with this background color (e.g. to highlight errors)

procedure DrawTextAt(s: SynUnicode; XPos: integer; const aLink: string='';
CheckPageNumber: boolean=false; aLinkNoBorder: boolean=false);

Draw one line of text, with the current alignment
procedure DrawTextFmt(const s: string; const Args: array of const; withNewLine: boolean=true);
  Draw some text as a paragraph, with the current alignment
  - this method use format() like parameters

procedure DrawTextU(const s: RawUTF8; withNewLine: boolean=true);
  Draw some UTF-8 text as a paragraph, with the current alignment
  - this method does all word-wrapping and formatting if necessary
  - this method handle multiple paragraphs inside s (separated by newlines - i.e. #13)
  - by default, will write a paragraph, unless withNewLine is set to FALSE, so that the next
    DrawText() will continue drawing at the current position

procedure DrawTextW(const s: SynUnicode; withNewLine: boolean=true);
  Draw some Unicode text as a paragraph, with the current alignment
  - this method does all word-wrapping and formatting if necessary
  - this method handle multiple paragraphs inside s (separated by newlines - i.e. #13)
  - by default, will write a paragraph, unless withNewLine is set to FALSE, so that the next
    DrawText() will continue drawing at the current position

procedure DrawTitle(const s: SynUnicode; DrawBottomLine: boolean=false;
  OutlineLevel: Integer=0; const aBookmark: string=''; const aLink: string='';
  aLinkNoBorder: boolean=false);
  Draw some text as a paragraph title
  - the outline level can be specified, if UseOutline property is enabled
  - if aBookmark is set, a bookmark is created at this position
  - if aLink is set, a link to the specified bookmark name (in aLink) is made

procedure EndDoc;
  End the Report document
  - Every report must start with BeginDoc and end with EndDoc

procedure EndGroup;
  End a previously defined Group
  - BeginGroup-EndGroup text blocks can't be nested

procedure GotoPosition(aPage: integer; aYPos: integer);
  Go to the specified Y position on a given page
  - used e.g. by GotoBookmark() method

procedure Invalidate; override;
  Customized invalidate

procedure NewHalfLine;
  Jump some half line space between paragraphs
  - Increments the current Y Position the equivalent of an half single line relative to the current
    font height and line spacing

procedure NewLine;
  Jump some line space between paragraphs
  - Increments the current Y Position the equivalent of a single line relative to the current font
    height and line spacing
procedure NewLines(count: integer);

*Jump some line space between paragraphs*
- Increments the current Y Position the equivalent of 'count' lines relative to the current font height and line spacing

procedure NewPage(ForceEndGroup: boolean=false);

*Jump to next page, i.e. force a page break*

procedure NewPageIfAnyContent;

*Jump to next page, but only if some content is pending*

procedure NewPageLayout(paperSize: TGdiPagePaperSize; orientation: TPrinterOrientation=poPortrait; nonPrintableWidthMM: integer=-1; nonPrintableHeightMM: integer=-1); overload;

*Change the page layout for the upcoming page*
- will then force a page break by a call to NewPage(true) method
- can change the default margin if margin*>=0
- can change the default non-printable printer margin if nonPrintable*>=0

procedure NewPageLayout(sizeWidthMM, sizeHeightMM: integer; nonPrintableWidthMM: integer=-1; nonPrintableHeightMM: integer=-1); overload;

*Change the page layout for the upcoming page*
- will then force a page break by a call to NewPage(true) method
- can change the default margin if margin*>=0
- can change the default non-printable printer margin if nonPrintable*>=0

procedure PopupMenuItemClick(Sender: TObject);

*This is the main popup menu item click event*

procedure RestoreSavedLayout; virtual;

*Restore last saved font and alignment*

procedure SaveLayout; virtual;

*Save the current font and alignment*

procedure SetColumnAlign(index: integer; align: TColAlign);

*Individually set column alignment*
- useful after having used AddColumns([]) method e.g.

procedure SetColumnBold(index: integer);

*Individually set column bold state*
- useful after having used AddColumns([]) method e.g.

procedure SetTabStops(const tabs: array of integer);

*Set the Tabs stops on every line*
- if one value is provided, it will set the Tabs as every multiple of it
- if more than one value are provided, they will be the exact Tabs positions

procedure ShowPreviewForm(VisibleButtons: TGdiPagePreviewButtons = [rNextPage..High(TGdiPagePreviewButtonClick)]);

*Show a form with the preview, allowing the user to browse pages and print the report*
- you can customize the buttons and popup menu actions displayed on the screen - by default, all buttons are visible
property BiDiMode: TBiDiMode read fBiDiMode write fBiDiMode;

Specifies the reading order (bidirectional mode) of the box
- only bdLeftToRight and bdRightToLeft are handled
- this will be used by DrawText[At], DrawTitle, AddTextToHeader/Footer[At],
  DrawTextAcrossCols, SaveLayout/RestoreSavedLayout methods

property Canvas: TMetaFileCanvas read fCanvas;

Can be used to draw directly using GDI commands
- The Canvas property should be rarely needed

property ColumnCount: integer read GetColumnCount;

Retrieve the current Column count

property CurrentYPos: integer read GetYPos write SetYPos;

Distance (in mm's) from the top of the page to the top of the next line

property ExportPDFA1: Boolean read fExportPDFA1 write fExportPDFA1;

If set to TRUE, the exported PDF is made compatible with PDF/A-1 requirements

property ExportPDFApplication: string read fExportPDFApplication write fExportPDFApplication;

Optional application name used during Export to PDF
- if not set, global Application.Title will be used

property ExportPDFAuthor: string read fExportPDFAuthor write fExportPDFAuthor;

Optional Author name used during Export to PDF

property ExportPDFBackground: TGraphic read fExportPDFBackground write fExportPDFBackground;

An optional background image, to be exported on every pdf page
- note that no private copy of the TGraphic instance is made: the caller has to manage it, and
  free it after the pdf is generated

property ExportPDFEmbeddedTTF: boolean read fExportPDFEmbeddedTTF write fExportPDFEmbeddedTTF;

If set to TRUE, the used True Type fonts will be embedded to the exported PDF
- not set by default, to save disk space and produce tiny PDF

property ExportPDFEncryptionLevel: TPdfEncryptionLevel read fExportPDFEncryptionLevel write fExportPDFEncryptionLevel;

Set encryption level to be used in exporting PDF document

property ExportPDFEncryptionOwnerPassword: string read fExportPDFEncryptionOwnerPassword write fExportPDFEncryptionOwnerPassword;

Set encryption owner password to be used in exporting PDF document
- it is mandatory to set it to a non void value - by default, is set to 'SynopsePDFEngine' by should
  be overridden for security
- ExportPDFEncryptionLevel = elRC4_40/elRC4_128 expects only ASCII-7 chars
property ExportPDFEncryptionPermissions: TPdfEncryptionPermissions read fExportPDFEncryptionPermissions write fExportPDFEncryptionPermissions;

Set encryption Permissions to be used in exporting PDF document
- can be either one of the PDF_PERMITION_ALL / PDF_PERMITION_NOMODIF / PDF_PERMITION_NOPRINT / PDF_PERMITION_NOCOPY / PDF_PERMITION_NOCOPYNORPRINT set of options
- default value is PDF_PERMITION_ALL (i.e. no restriction)

property ExportPDFEncryptionUserPassword: string read fExportPDFEncryptionUserPassword write fExportPDFEncryptionUserPassword;

Set encryption user password to be used in exporting PDF document
- leave it to " unless you want the user to be asked for this password at document opening
- ExportPDFEncryptionLevel = elRC4_40/elRC4_128 expects only ASCII-7 chars

property ExportPDFFontFallBackName: string read fExportPDFFontFallBackName write fExportPDFFontFallBackName;

Set the font name to be used for missing characters in exported PDF document
- used only if UseFontFallBack is TRUE
- default value is 'Arial Unicode MS', if existing

property ExportPDFForceJPEGCompression: integer read fForceJPEGCompression write fForceJPEGCompression;

This property can force saving all bitmaps as JPEG in exported PDF
- by default, this property is set to 0 by the constructor of this class, meaning that the JPEG compression is not forced, and the engine will use the native resolution of the bitmap - in this case, the resulting PDF file content will be bigger in size (e.g. use this for printing)
- 60 is the preferred way e.g. for publishing PDF over the internet
- 80/90 is a good ration if you want to have a nice PDF to see on screen
- of course, this doesn’t affect vectorial (i.e. emf) pictures

property ExportPDFGeneratePDF15File: Boolean read fExportPDFGeneratePDF15File write fExportPDFGeneratePDF15File;

Set to TRUE to export in PDF 1.5 format, which may produce smaller files

property ExportPDFKeywords: string read fExportPDFKeywords write fExportPDFKeywords;

Optional Keywords name used during Export to PDF

property ExportPDFSubject: string read fExportPDFSubject write fExportPDFSubject;

Optional Subject text used during Export to PDF

property ExportPDFUseFontFallBack: boolean read fExportPDFUseFontFallBack write fExportPDFUseFontFallBack;

Used to define if the exported PDF document will handle "font fallback" for characters not existing in the current font: it will avoid rendering block/square symbols instead of the correct characters (e.g. for Chinese text)
- will use the font specified by FontFallBackName property to add any Unicode glyph not existing in the currently selected font
- default value is TRUE
property ExportPDFUseUniscribe: boolean read fExportPDFUseUniscribe write fExportPDFUseUniscribe;
    Set if the exporting PDF engine must use the Windows Uniscribe API to render Ordering and/or Shaping of the text
    - useful for Hebrew, Arabic and some Asiatic languages handling
    - set to FALSE by default, for faster content generation

property ForceScreenResolution: boolean read fForceScreenResolution write fForceScreenResolution;
    If set to true, we reduce the precision for better screen display

property HangIndent: integer read fHangIndent write fHangIndent;
    Left justification hang indentation

property HeaderDone: boolean read fHeaderDone;
    True if any header as been drawn, that is if something is to be printed

property LeftMargin: integer read GetLeftMargin write SetLeftMargin;
    Size of the left margin relative to its corresponding edge in mm's

property LineHeight: integer read GetLineHeightMm;
    Get current line height (mm)

property LineSpacing: TLineSpacing read fLineSpacing write fLineSpacing;
    Line spacing: can be lsSingle, lsOneAndHalf or lsDouble

property NegsToParenthesesInCurrCols: boolean read fNegsToParenthesesInCurrCols write fNegsToParenthesesInCurrCols;
    Accounting standard layout for caCurrency columns:
    - convert all negative sign into parentheses
    - using parentheses instead of negative numbers is used in financial statement reporting (see e.g. http://en.wikipedia.org/wiki/Income_statement)
    - align numbers on digits, not parentheses

    Event triggered whenever the report document generation is done
    - i.e. when the EndDoc method has just been called

property OnEndColumnHeader: TNotifyEvent read fEndColumnHeader write fEndColumnHeader;
    Event triggered when each column was drawn

property OnEndPageFooter: TNotifyEvent read fEndPageFooter write fEndPageFooter;
    Event triggered when each footer was drawn

property OnEndPageHeader: TNotifyEvent read fEndPageHeader write fEndPageHeader;
    Event triggered when each header was drawn

    Event triggered when each new page is created

property OnPreviewPageChanged: TNotifyEvent read fPreviewPageChangedEvent write fPreviewPageChangedEvent;
    Event triggered whenever the current preview page is changed
property OnStartColumnHeader: TNotifyEvent read fStartColumnHeader write fStartColumnHeader;

  Event triggered when each new column is about to be drawn

property OnStartPageFooter: TNotifyEvent read fStartPageFooter write fStartPageFooter;

  Event triggered when each new footer is about to be drawn

property OnStartPageHeader: TNotifyEvent read fStartPageHeader write fStartPageHeader;

  Event triggered when each new header is about to be drawn

property OnZoomChanged: TZoomChangedEvent read fZoomChangedEvent write fZoomChangedEvent;

  Event triggered whenever the preview page is zoomed in or out

property Orientation: TPrinterOrientation read GetOrientation write SetOrientation;

  The paper orientation

property Page: integer read fCurrPreviewPage write SetPage;

  The index of the previewed page
  - please note that the first page is 1 (not 0)

property PageCount: integer read GetPageCount;

  Total number of pages

property PageMargins: TRect read GetPageMargins write SetPageMargins;

  Size of each margin relative to its corresponding edge in mm's

property Pages: TGDIPageContentDynArray read fPages;

  Access to all pages content
  - numerotation begin with Pages[0] for page 1
  - the Pages[] property should be rarely needed

property PaperSize: TSize read GetPaperSize;

  Get the current selected paper size, in mm's

property PrinterName: string read fCurrentPrinter;

  The name of the current selected printer
  - note that Printers.SetPrinter() should be set BEFORE calling BeginDoc, otherwise you may have a "canvas does not allow drawing" error

property PrinterPxPerInch: TPoint read fPrinterPxPerInch;

  Number of pixel per inch, for X and Y directions

property RightMarginPos: integer read GetRightMarginPos;

  Position of the right margin, in mm

property TextAlign: TTextAlign read fAlign write SetTextAlign;

  The current Text Alignment, during text adding

property UseOutlines: boolean read fUseOutlines write fUseOutlines;

  If set, any DrawTitle() call will create an Outline entry
  - used e.g. for PDF generation
  - this is enabled by default
property VirtualPageNum: integer read fVirtualPageNum write fVirtualPageNum;
  The current page number, during text adding
  - Page is used during preview, after text adding

property WordWrapLeftCols: boolean read fWordWrapLeftCols write fWordWrapLeftCols;
  Word wrap (caLeft) left-aligned columns into multiple lines
  - if the text is wider than the column width, its content is wrapped to the next line
  - if the text contains some #13/#10 characters, it will be splitted into individual lines
  - this is disabled by default

property Zoom: integer read fZoom write SetZoom;
  The current Zoom value, according to the zoom status
  - you can use PAGE_WIDTH and PAGE_FIT constants to force the corresponding zooming mode
    (similar to ZoomStatus property setter)
  - set this property will work only when the report is already shown in preview mode, not before
    ShowPreviewForm method call

property ZoomStatus: TZoomStatus read fZoomStatus write SetZoomStatus;
  The current Zoom procedure, i.e. zsPercent, zsPageFit or zsPageWidth
  - set this property will define the Zoom at PAGE_WIDTH or PAGE_FIT special constant, if needed
  - set this property will work only when the report is already shown in preview mode, not before
    ShowPreviewForm method call

Types implemented in the mORMotReport unit

TColAlign = ( caLeft, caRight, caCenter, caCurrency );
  Text column alignment

TGDIPageContentDynArray = array of TGDIPageContent;
  Used to store all pages of the report

TGdiPagePaperSize = ( psA4, psA5, psA3, psLetter, psLegal );
  Available known paper size for NewPageLayout() method

TGdiPagePreviewButton = ( rNone, rNextPage, rPreviousPage, rGotoPage, rZoom,
  rBookmarks, rPageAsText, rPrint, rExportPDF, rClose );
  The available menu items

TGdiPagePreviewButtons = set of TGdiPagePreviewButton;
  Set of menu items

TLineSpacing = ( lsSingle, lsOneAndHalf, lsDouble );
  Text line spacing

TNewPageEvent = procedure(Sender: TObject; PageNumber: integer) of object;
  Event triggered when a new page is added

TOnStringToUnicodeEvent = function(const Text: SynUnicode): SynUnicode of object;
  Event triggered to allow custom unicode character display on the screen
  - called for all text, whatever the alignment is
  - Text content can be modified by this event handler to customize some characters (e.g. '=>' can be
    converted to the one Unicode glyph)

TTTextAlign = ( taLeft, taRight, taCenter, taJustified );
Text paragraph alignment

TZoomChangedEvent = procedure(Sender: TObject; Zoom: integer; ZoomStatus: TZoomStatus) of object;

Event triggered when the Zoom was changed

TZoomStatus = ( zsPercent, zsPageFit, zsPageWidth );

Available zoom mode
- zsPercent is used with a zoom percentage (e.g. 100% or 50%)
- zsPageFit fits the page to the report
- zsPageWidth zooms the page to fit the report width on screen

Constants implemented in the mORMotReport unit

FORMAT_ALIGN_MASK = $300;
  Alignment bits 8-9

FORMAT_BOLD = $400;
  Fontstyle bits 10-12

FORMAT_DEFAULT = $0;
  TEXT FORMAT FLAGS...

FORMAT_SINGELELINE = $8000;
  Line flags bits 14-15

FORMAT_SIZE_MASK = $FF;
  Fontsize bits 0-7 . max = 255

FORMAT_UNDEFINED = $2000;
  Undefined bit 13

FORMAT_XPOS_MASK = $FFFF0000;
  DrawTextAt XPos 16-30 bits (max value = ~64000)

GRAY_MARGIN = 10;
  Minimum gray border with around preview page

PAGENUMBER = '<<pagenumber>>';
  This constant can be used to be replaced by the page number in the middle of any text

PAGE_FIT = -2;
  TGdiPages.Zoom property value for "Page fit" layout during preview

PAGE_WIDTH = -1;
  TGdiPages.Zoom property value for "Page width" layout during preview
27.63. mORMotSelfTests.pas unit

*Purpose*: Automated tests for common units of the Synopse mORMot Framework
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the mORMotSelfTests unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynBidirSock</td>
<td>Implements bidirectional client and server protocol, e.g. WebSockets</td>
<td>680</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynBigTable</td>
<td>Class used to store huge amount of data with fast retrieval</td>
<td>704</td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynSelfTests</td>
<td>Automated tests for common units of the Synopse mORMot Framework</td>
<td>1528</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynSQLite3</td>
<td>SQLite3 Database engine direct access</td>
<td>1646</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTests</td>
<td>Unit test functions used by Synopse projects</td>
<td>1832</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

**Functions or procedures implemented in the mORMotSelfTests unit**

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite3ConsoleTests</td>
<td></td>
<td>2369</td>
</tr>
</tbody>
</table>

**procedure** SQLite3ConsoleTests;

*This is the main entry point of the tests*
- this procedure will create a console, then run all available tests
27.64. mORMotService.pas unit

Purpose: Daemon management classes for mORMot, including low-level Win NT Service
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the mORMotService unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1899</td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher) - implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT - optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support) - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1139</td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1363</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features - as a complement to SynCommons, which tended to increase too much - licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1721</td>
</tr>
</tbody>
</table>

Objects implemented in the mORMotService unit

```
TObject
  \-- TSynWindowsPrivileges
  \-- TserviceStatus
  \-- TServiceStatusProcess
    \-- TServiceController
    \-- TService
      \-- TSynDaemonSettings
      \-- TSynPersistent
    \-- TSynJsonFileSettings
    \-- TSynDaemon
```

mORMotService class hierarchy
## Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TService</td>
<td>TService is the class used to implement a service provided by an application</td>
<td>2374</td>
</tr>
<tr>
<td>TServiceController</td>
<td>TServiceController class is intended to create a new service instance or to maintain (that is start, stop, pause, resume...) an existing service</td>
<td>2371</td>
</tr>
<tr>
<td>TServiceSingle</td>
<td>Inherit from this service if your application has a single service</td>
<td>2376</td>
</tr>
<tr>
<td>TSynDaemon</td>
<td>Abstract parent to implements a daemon/service</td>
<td>2377</td>
</tr>
<tr>
<td>TSynDaemonSettings</td>
<td>Abstract parent containing information able to initialize a TSynDaemon class</td>
<td>2377</td>
</tr>
<tr>
<td>TSynWindowsPrivileges</td>
<td>Object dedicated to management of available privileges for Windows platform</td>
<td>2378</td>
</tr>
</tbody>
</table>

**TServiceController = class(TObject)**

- *TServiceController class is intended to create a new service instance or to maintain (that is start, stop, pause, resume...) an existing service*
- To provide the service itself, use the TService class
constructor CreateNewService(const TargetComputer, DatabaseName, Name, DisplayName, Path: string; const OrderGroup: string = ''; const Dependencies: string = ''; const Username: string = ''; const Password: string = ''; DesiredAccess: DWORD = SERVICE_ALL_ACCESS; ServiceType: DWORD = SERVICE_WIN32_OWN_PROCESS or SERVICE_INTERACTIVE_PROCESS; StartType: DWORD = SERVICE_DEMAND_START; ErrorControl: DWORD = SERVICE_ERROR_NORMAL);

Creates a new service and allows to control it and/or its configuration

- TargetComputer - set it to empty string if local computer is the target.
- DatabaseName - set it to empty string if the default database is supposed ('ServicesActive').
- Name - name of a service.
- DisplayName - display name of a service.
- Path - a path to binary (executable) of the service created.
- OrderGroup - an order group name (unnecessary)
- Dependencies - string containing a list with names of services, which must start before (every name should be separated with #0, entire list should be separated with #0#0. Or, an empty string can be passed if there is no dependency).
- Username - login name. For service type SERVICE_WIN32_OWN_PROCESS, the account name in the form of "DomainName\Username"; If the account belongs to the built-in domain, ",\Username" can be specified; Services of type SERVICE_WIN32_SHARE_PROCESS are not allowed to specify an account other than LocalSystem. If " is specified, the service will be logged on as the 'LocalSystem' account, in which case, the Password parameter must be empty too.
- Password - a password for login name. If the service type is SERVICE_KERNEL_DRIVER or SERVICE_FILE_SYSTEM_DRIVER, this parameter is ignored.
- DesiredAccess - a combination of following flags: SERVICE_ALL_ACCESS (default value), SERVICE_CHANGE_CONFIG, SERVICE_ENUMERATE_DEPENDENTS, SERVICE_INTERROGATE, SERVICE_PAUSE_CONTINUE, SERVICE_QUERY_CONFIG, SERVICE_QUERY_STATUS, SERVICE_START, SERVICE_STOP, SERVICE_USER_DEFINED_CONTROL
- ServiceType - a set of following flags: SERVICE_WIN32_OWN_PROCESS (default value, which specifies a Win32 service that runs in its own process), SERVICE_WIN32_SHARE_PROCESS, SERVICE_KERNEL_DRIVER, SERVICE_FILE_SYSTEM_DRIVER, SERVICE_INTERACTIVE_PROCESS (default value, which enables a Win32 service process to interact with the desktop)
- StartType - one of following values: SERVICE_BOOT_START, SERVICE_SYSTEM_START, SERVICE_AUTO_START (which specifies a device driver or service started by the service control manager automatically during system startup), SERVICE_DEMAND_START (default value, which specifies a service started by a service control manager when a process calls the StartService function, that is the TServiceController.Start method), SERVICE_DISABLED
- ErrorControl - one of following: SERVICE_ERROR_IGNORE, SERVICE_ERROR_NORMAL (default value, by which the startup program logs the error and displays a message but continues the startup operation), SERVICE_ERROR_SEVERE, SERVICE_ERROR_CRITICAL
constructor CreateOpenService(const TargetComputer, DataBaseName, Name: String; DesiredAccess: DWORD = SERVICE_ALL_ACCESS);

Opens an existing service, in order to control it or its configuration from your application.
Parameters (strings are unicode-ready since Delphi 2009):
- TargetComputer - set it to empty string if local computer is the target.
- DatabaseName - set it to empty string if the default database is supposed ('ServicesActive').
- Name - name of a service.
- DesiredAccess - a combination of following flags: SERVICE_ALL_ACCESS,
  SERVICE_CHANGE_CONFIG, SERVICE_ENUMERATE_DEPENDENTS, SERVICE_INTERROGATE,
  SERVICE_PAUSE_CONTINUE, SERVICE_QUERY_CONFIG, SERVICE_QUERY_STATUS,
  SERVICE_START, SERVICE_STOP, SERVICE_USER_DEFINED_CONTROL

destructor Destroy; override;

Release memory and handles

function Delete: boolean;

Removes service from the system, i.e. close the Service

class function Install(const Name, DisplayName, Description: string; AutoStart: boolean; ExeName: TFileName=''; Dependencies: string=''): TServiceState;

Wrapper around CreateNewService() to install the current executable as service

function Pause: boolean;

Requests the service to pause

function Refresh: boolean;

Requests the service to update immediately its current status information to the service control manager

function Resume: boolean;

Requests the paused service to resume

function Shutdown: boolean;

Request the service to shutdown
- this function always return false

function Start(const Args: array of PChar): boolean;

Starts the execution of a service with some specified arguments
- this version expect PChar pointers, either AnsiString (for FPC and old Delphi compiler), either
  UnicodeString (till Delphi 2009)

function Stop: boolean;

Requests the service to stop
class procedure CheckParameters(const ExeFileName: TFileName; const ServiceName, DisplayName, Description: string; const Dependencies: string='');

This class method will check the command line parameters, and will let control the service according to it
- MyServiceSetup.exe /install will install the service
- MyServiceSetup.exe /start will start the service
- MyServiceSetup.exe /stop will stop the service
- MyServiceSetup.exe /uninstall will uninstall the service
- so that you can write in the main block of your .dpr:
  CheckParameters('MyService.exe', HTTPSERVICENAME, HTTPSERVICEDisplayNAME);
- if ExeFileName='', it will install the current executable
- optional Description and Dependencies text may be specified

procedure SetDescription(const Description: string);

Try to define the description text of this service

property Handle: THandle read FHandle;
Handle of service opened or created
- its value is 0 if something failed in any Create*() method

property SCHandle: THandle read FSCHandle;
Handle of SC manager

property State: TServiceState read GetState;
Retrieve the Current state of the service

property Status: TServiceStatus read GetStatus;
Retrieve the Current status of the service

TService = class(TObject)

TService is the class used to implement a service provided by an application

constructor Create(const aServiceName, aDisplayName: String); reintroduce; virtual;
Creates the service
- the service is added to the internal registered services
- main application must call the global ServicesRun procedure to actually start the services
- caller must free the TService instance when it's no longer used

destructor Destroy; override;
Free memory and release handles

function Install(const Params: string=''): boolean;
Installs the service in the database
- return true on success
- create a local TServiceController with the current executable file, with the supplied command line parameters

function ReportStatus(dwState, dwExitCode, dwWait: DWORD): BOOL;
Reports new status to the system
procedure DoCtrlHandle(Code: DWORD); virtual;

    This method is the main service entrance, from the OS point of view
    - it will call OnControl/OnStop/OnPause/OnResume/OnShutdown events
    - and report the service status to the system (via ReportStatus method)

procedure Execute; virtual;

    This is the main method, in which the Service should implement its run

procedure Remove;

    Removes the service from database
    - uses a local TServiceController with the current Service Name

procedure Start;

    Starts the service
    - uses a local TServiceController with the current Service Name

procedure Stop;

    Stops the service
    - uses a local TServiceController with the current Service Name

property ArgCount: Integer read GetArgCount;

    Number of arguments passed to the service by the service controller

property Args[Idx: Integer]: String read GetArgs;

    List of arguments passed to the service by the service controller

property ControlHandler: TServiceControlHandler read GetControlHandler write SetControlHandler;

    Callback handler for Windows Service Controller
    - if handler is not set, then auto generated handler calls DoCtrlHandle (note that this
      auto-generated stubb is... not working yet - so you should either set your own procedure to this
      property, or use TServiceSingle)
    - a typical control handler may be defined as such:
      var MyGlobalService: TService;

      procedure MyServiceControlHandler(Opcode: LongWord);stdcall;
      begin
        if MyGlobalService<>nil then
          MyGlobalService.DoCtrlHandle(Opcode);
      end;
      ...
      MyGlobalService := TService.Create(...
      MyGlobalService.ControlHandler := MyServiceControlHandler;

property Data: DWORD read FData write FData;

    Any data You wish to associate with the service object

property DisplayName: String read fName write fName;

    Display name of the service

property Installed: boolean read GetInstalled;

    Whether service is installed in DataBase
    - uses a local TServiceController to check if the current Service Name exists
property OnControl: TServiceControlEvent read fOnControl write fOnControl;
  Custom event triggered when a Control Code is received from Windows

property OnExecute: TServiceEvent read fOnExecute write fOnExecute;
  Custom Execute event
  - launched in the main service thread (i.e. in the Execute method)

property OnInterrogate: TServiceEvent read fOnInterrogate write fOnInterrogate;
  Custom event triggered when the service receive an Interrogate

property OnPause: TServiceEvent read fOnPause write fOnPause;
  Custom event triggered when the service is paused

property OnResume: TServiceEvent read fOnResume write fOnResume;
  Custom event triggered when the service is resumed

property OnShutdown: TServiceEvent read fOnShutdown write fOnShutdown;
  Custom event triggered when the service is shut down

property OnStart: TServiceEvent read fOnStart write fOnStart;
  Start event is executed before the main service thread (i.e. in the Execute method)

property OnStop: TServiceEvent read fOnStop write fOnStop;
  Custom event triggered when the service is stopped

property ServiceName: String read fSName;
  Name of the service. Must be unique

property ServiceType: DWORD read fServiceType write fServiceType;
  Type of service

property StartType: DWORD read fStartType write fStartType;
  Type of start of service

property Status: TServiceStatus read fStatusRec write SetStatus;
  Current service status
  - To report new status to the system, assign another value to this record, or use ReportStatus
  method (preferred)

TServiceSingle = class(TService)

  Inherit from this service if your application has a single service
  - note that TService jumper does not work well - so use this instead

  constructor Create(const aServiceName, aDisplayName: String); override;
    Will set a global function as service controller

  destructor Destroy; override;
    Will release the global service controller
TSynDaemonSettings = class(TSynJsonFileSettings)
  Abstract parent containing information able to initialize a TSynDaemon class
  - will handle persistence as JSON local files
  - you may consider using TDDDAppSettingsAbstract from dddInfraSettings

constructor Create; override;
  Initialize and set the default settings

function ServiceDescription: string;
  Returns user-friendly description of the service, including version information and company copyright (if available)

procedure SetLog(aLogClass: TSynLogClass);
  Define the log information into the supplied TSynLog class
  - if you don't call this method, the logging won't be initiated
  - is to be called typically in the overridden Create constructor of the associated TSynDaemon class, just after "inherited Create"

property Log: TSynLogInfos read fLog write fLog;
  If not void, will enable the logs (default is LOG_STACKTRACE)

property LogClass: TSynLogClass read fLogClass;
  Read-only access to the TSynLog class, if SetLog() has been called

property LogPath: TFileName read fLogPath write fLogPath;
  Allow to customize where the logs should be written

property LogRotateFileCount: integer read fLogRotateFileCount write fLogRotateFileCount;
  How many files will be rotated (default is 2)

property ServiceDependencies: string read fServiceDependencies write fServiceDependencies;
  Optional service dependencies
  - not published by default: could be defined if needed, or e.g. set in overridden constructor
  - several depending services may be set by appending #0 between names

property ServiceDisplayName: string read fServiceDisplayName write fServiceDisplayName;
  The service name, as displayed by Windows or at the console level
  - default is the executable name

property ServiceName: string read fServiceName write fServiceName;
  The service name, as used internally by Windows or the TSynDaemon class
  - default is the executable name

TSynDaemon = class(TSynPersistent)
  Abstract parent to implements a daemon/service
  - inherit from this abstract class and override Start and Stop methods
  - you may consider using TDDDAdministratedDaemon from dddInfraApps
constructor Create(aSettingsClass: TSynDaemonSettingsClass; const aWorkFolder, aSettingsFolder, aLogFolder: TFileName; const aSettingsExt: TFileName = '.settings'; const aSettingsName: TFileName = ''); reintroduce;

Initialize the daemon, creating the associated settings
- TSynDaemonSettings instance will be owned and freed by the daemon
- any non supplied folder name will be replaced by a default value (executable folder under Windows, or /etc /var/log on Linux)

destructor Destroy; override;

Call Stop, finalize the instance, and its settings

procedure CommandLine(aAutoStart: boolean=true);
Main entry point of the daemon, to process the command line switches
- aAutoStart is used only under Windows

procedure Start; virtual; abstract;
Inherited class should override this abstract method with proper process

procedure Stop; virtual; abstract;
Inherited class should override this abstract method with proper process
- should do nothing if the daemon was already stopped

property ConsoleMode: boolean read fConsoleMode;
If this instance was run as /console or /verb

property Settings: TSynDaemonSettings read fSettings;
The settings associated with this daemon
- will be allocated in Create constructor, and released in Destroy

TSynWindowsPrivileges = object(TObject)
Object dedicated to management of available privileges for Windows platform
- not all available privileges are active for process
- for usage of more advanced WinAPI, explicit enabling of privilege is sometimes needed

Token: THandle;
Handle to privileges token

function Disable(aPrivilege: TWinSystemPrivilege): boolean;
Disable privilege
- if aPrivilege is already disabled return true, if operation is not possible (required privilege doesn't exist or API error) return false

function Enable(aPrivilege: TWinSystemPrivilege): boolean;
Enable privilege
- if aPrivilege is already enabled return true, if operation is not possible (required privilege doesn't exist or API error) return false

procedure Done(aRestoreInitiallyEnabled: boolean = true);
Finalize the object and release Token handle
- aRestoreInitiallyEnabled parameter can be used to restore initially state of enabled privileges
procedure Init(aTokenPrivilege: TPrivilegeTokenType = pttProcess);

- Initialize the object dedicated to management of available privileges
  - aTokenPrivilege can be used for current process or current thread

property Available: TWinSystemPrivileges read fAvailable;

- Set of available privileges for current process/thread

property Enabled: TWinSystemPrivileges read fEnabled;

- Set of enabled privileges for current process/thread

Types implemented in the mORMotService unit

TParseCommand = ( pcHasRedirection, pcHasSubCommand, pcHasParenthesis,
  pcHasJobControl, pcHasShellVariable, pcUnbalancedSingleQuote, pcUnbalancedDoubleQuote, pcTooManyArguments, pcInvalidCommand, pcHasEndingBackSlash);

- Command line patterns recognized by ParseCommandArgs()

TParseCommandsArgs = array[0..31] of PAnsiChar;

- Used to store references of arguments recognized by ParseCommandArgs()

TServiceControlEvent = procedure(Sender: TService; Code: DWORD) of object;

- Event triggered for Control handler

TServiceControlHandler = procedure(CtrlCode: DWORD); stdcall;

- Callback procedure for Windows Service Controller

TServiceEvent = procedure(Sender: TService) of object;

- Event triggered to implement the Service functionality

TServiceState = ( ssNotInstalled, ssStopped, ssStarting, ssStopping, ssRunning,
  ssResuming, ssPausing, ssPaused, ssErrorRetrievingState );

- All possible states of the service

TSynDaemonSettingsClass = class of TSynDaemonSettings;

- Meta-class of TSynDaemon settings information

TWinSystemPrivilege = ( wspCreateToken, wspAssignPrimaryToken, wspLockMemory,
  wspIncreaseQuota, wspUnsolicitedInput, wspMachineAccount, wspTCP, wspSecurity,
  wspTakeOwnership, wspLoadDriver, wspSystemProfile, wspSystemTime,
  wspProfSingleProcess, wspIncBasePriority, wspCreatePageFile, wspCreatePermanent,
  wspBackup, wspRestore, wspShutdown, wspDebug, wspAudit, wspSystemEnvironment,
  wspChangeNotify, wspRemoteShutdown, wspUndock, wspSyncAgent, wspEnableDelegation,
  wspManageVolume, wspImpersonate, wspCreateGlobal, wspTrustedCredmanAccess,
  wspRelabel, wspIncWorkingSet, wspTimeZone, wspCreateSymbolicLink );

- Enum synchronized with WinAPI
  - see https://docs.microsoft.com/en-us/windows/desktop/secauthz/privilege-constants

Constants implemented in the mORMotService unit

CMDOINESWITCH = '/';

- Text identifier typically used before command line switches
  - equals '/' on Windows, and '--' on POSIX systems

PARSECOMMAND_BASH = [pcHasRedirection .. pcHasShellVariable];
Identifies some bash-specific processing

PARSECOMMAND_ERROR = [pcUnbalancedSingleQuote .. pcHasEndingBackSlash];

Identifies obvious invalid content

## Functions or procedures implemented in the mORMotService unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrentStateToServiceState</td>
<td>Convert the Control Code retrieved from Windows into a service state enumeration item</td>
<td>2380</td>
</tr>
<tr>
<td>GetServicePid</td>
<td>Return service PID</td>
<td>2380</td>
</tr>
<tr>
<td>KillProcess</td>
<td>Kill Windows process</td>
<td>2380</td>
</tr>
<tr>
<td>ParseCommandArgs</td>
<td>Low-level parsing of a RunCommand() execution command</td>
<td>2380</td>
</tr>
<tr>
<td>RunCommand</td>
<td>Like fpSystem, but cross-platform</td>
<td>2380</td>
</tr>
<tr>
<td>RunProcess</td>
<td>Like SysUtils.ExecuteProcess, but allowing not to wait for the process to finish</td>
<td>2381</td>
</tr>
<tr>
<td>ServicesRun</td>
<td>Launch the registered Services execution</td>
<td>2381</td>
</tr>
<tr>
<td>ServiceStateText</td>
<td>Return the ready to be displayed text of a TServiceState value</td>
<td>2381</td>
</tr>
</tbody>
</table>

**function** CurrentStateToServiceState(CurrentState: DWORD): TServiceState;

Convert the Control Code retrieved from Windows into a service state enumeration item

**function** GetServicePid(const aServiceName: string): DWORD;

Return service PID

**function** KillProcess(pid: DWORD; waitseconds: integer = 30): boolean;

Kill Windows process

**function** ParseCommandArgs(const cmd: RawUTF8; argv: PParseCommandsArgs = nil; argc: PInteger = nil; temp: PRawUTF8 = nil; posix: boolean = false): TParseCommands;

Low-level parsing of a RunCommand() execution command
- parse and fills argv^[0..argc^]-1 with corresponding arguments, after un-escaping and un-quoteding if applicable, using temp^ to store the content
- if argv=nil, do only the parsing, not the argument extraction - could be used for fast validation of the command line syntax
- you can force arguments OS flavor using the posix parameter - note that Windows parsing is not consistent by itself (e.g. double quoting or escaping depends on the actual executable called) so returned flags should be considered as indicative only with posix=false

**function** RunCommand(const cmd: TFileName; waitfor: boolean; const env: TFileName=''; envvaddexisting: boolean=false; parsed: PParseCommands=nil): integer;

Like fpSystem, but cross-platform
- under POSIX, calls bash only if needed, after ParseCommandArgs() analysis
- under Windows (especially Windows 10), creating a process can be dead slow

https://randomascii.wordpress.com/2019/04/21/on2-in-createprocess
**function** RunProcess(const path, arg1: TFileName; waitfor: boolean; const arg2: TFileName=''; const arg3: TFileName=''; const arg4: TFileName=''; const arg5: TFileName=''; const env: TFileName=''; envaddexisting: boolean=false): integer;

Like SysUtils.ExecuteProcess, but allowing not to wait for the process to finish
- optional env value follows 'n1=v1#0'n2=v2#0'n3=v3#0#0 Windows layout

**function** ServicesRun: boolean;

Launch the registered Services execution
- the registered list of service provided by the application is sent to the operating system
- returns TRUE on success
- returns FALSE on error (to get extended information, call GetLastError)

**function** ServiceStateText(State: TServiceState): string;

Return the ready to be displayed text of a TServiceState value

**Variables implemented in the mORMotService unit**

ServiceLog: TSynLogClass;

- you can set this global variable to TSynLog or TSQLLog to enable logging
- default is nil, i.e. disabling logging, since it may interfere with the logging process of the service itself

Services: TSynList = nil;

- the internal list of Services handled by this unit
- not to be accessed directly: create TService instances, and they will be added/registered to this list
- then run the global ServicesRun procedure
- every TService instance is to be freed by the main application, when it's no more used

ServiceSingle: TServiceSingle = nil;

The main TService instance running
27.65. mORMotSQLite3.pas unit

**Purpose:** SQLite3 embedded Database engine used as the mORMot SQL kernel
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The **mORMotSQLite3** unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.2.1</td>
<td>The SQLite3 engine shall be embedded to the framework</td>
<td>2548</td>
</tr>
</tbody>
</table>

Units used in the **mORMotSQLite3** unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mORMot</strong></td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><strong>SynCommons</strong></td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><strong>SynLog</strong></td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><strong>SynSQLite3</strong></td>
<td>SQLite3 Database engine direct access</td>
<td>1646</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><strong>SynTable</strong></td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><strong>SynZip</strong></td>
<td>Low-level access to ZLib compression (1.2.5 engine version)</td>
<td>1845</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

![mORMotSQLite3 class hierarchy](image)

**Objects implemented in the mORMotSQLite3 unit**
### Objects and Description

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLRestClientDB</td>
<td>REST client with direct access to a SQLite3 database</td>
<td>2387</td>
</tr>
<tr>
<td>TSQLRestServerDB</td>
<td>REST server with direct access to a SQLite3 database</td>
<td>2383</td>
</tr>
<tr>
<td>TSQLRestStorageShardedDB</td>
<td>REST storage sharded over several SQLite3 instances</td>
<td>2388</td>
</tr>
<tr>
<td>TSQLTableDB</td>
<td>Execute a SQL statement in the local SQLite3 database engine, and get result in memory</td>
<td>2383</td>
</tr>
<tr>
<td>TSQLVirtualTableModuleServerDB</td>
<td>Define a Virtual Table module for a TSQLRestServerDB SQLite3 engine</td>
<td>2388</td>
</tr>
<tr>
<td>TSQLVirtualTableModuleSQLite3</td>
<td>Define a Virtual Table module for a stand-alone SQLite3 engine</td>
<td>2388</td>
</tr>
</tbody>
</table>

**TSQLTableDB = class(TSQLTableJSON)**

*Execute a SQL statement in the local SQLite3 database engine, and get result in memory*
- all DATA (even the BLOB fields) is converted into UTF-8 TEXT
- uses a TSQLTableJSON internally: faster than sqlite3_get_table() (less memory allocation/fragmentation) and allows efficient caching

**constructor** Create(aDB: TSQLDatabase; const Tables: array of TSQLRecordClass; const aSQL: RawUTF8; Expand: boolean); reintroduce;

*Execute a SQL statement, and init TSQLTable fields*
- FieldCount=0 if no result is returned
- the BLOB data is converted into TEXT: you have to retrieve it with a special request explicitly (note that JSON format returns BLOB data)
- uses a TSQLTableJSON internally: all currency is transformed to its floating point TEXT representation, and allows efficient caching
- if the SQL statement is in the DB cache, it's retrieved from its cached value: our JSON parsing is a lot faster than SQLite3 engine itself, and uses less memory
- will raise an ESQLException on any error

**TSQLRestServerDB = class(TSQLRestServer)**

*REST server with direct access to a SQLite3 database*
- caching is handled at TSQLDatabase level
- SQL statements for record retrieval from ID are prepared for speed

*Used for DI-2.2.1 (page 2548).*

**constructor** Create(aModel: TSQLModel; const aDBFileName: TFileName; aHandleUserAuthentication: boolean=false; const aPassword: RawUTF8=''; aDefaultCacheSize: integer=10000; aDefaultPageSize: integer=4096); reintroduce; overload;

*Initialize a REST server with a database, by specifying its filename*
- TSQLRestServerDB will initialize a owned TSQLDatabase, and free it on Destroy
- if specified, the password will be used to cypher this file on disk (the main SQLite3 database file is encrypted, not the wal file during run)
- it will then call the other overloaded constructor to initialize the server
constructor Create(aModel: TSQLModel; aDB: TSQLDataBase; aHandleUserAuthentication: boolean=false; aOwnDB: boolean=false); reintroduce; overload; virtual;

Initialize a REST server with a SQLite3 database
- any needed TSQLVirtualTable class should have been already registered via the RegisterVirtualTableModule() method

constructor Create(aModel: TSQLModel; aHandleUserAuthentication: boolean=false); overload; override;

Initialize a REST server with an in-memory SQLite3 database
- could be used for test purposes

constructor CreateWithOwnModel(const aTables: array of TSQLRecordClass; aHandleUserAuthentication: boolean=false); overload;

Initialize a REST server with an in-memory SQLite3 database and a temporary Database Model
- could be used for test purposes

constructor CreateWithOwnModel(const aTables: array of TSQLRecordClass; const aDBFileName: TFileName; aHandleUserAuthentication: boolean=false; const aRoot: RawUTF8='root'; const aPassword: RawUTF8=''; aDefaultCacheSize: integer=10000; aDefaultPageSize: integer=4096); overload;

Initialize a REST server with a database, and a temporary Database Model
- a Model will be created with supplied tables, and owned by the server
- if you instantiate a TSQLRestServerFullMemory or TSQLRestServerDB with this constructor, an in-memory engine will be created, with enough abilities to run regression tests, for instance

destructor Destroy; override;

Close database and free used memory

function Backup(Dest: TStream): boolean; deprecated;

Backup of the opened Database into an external stream (e.g. a file, compressed or not)
- DEPRECATED: use DB.BackupBackground() instead
- this method doesn't use the SQLite Online Backup API, but low-level database file copy which may lock the database process if the data is consistent - consider using DB.BackupBackground() method instead
- database is closed, VACCUUMed, copied, then reopened

function BackupGZ(const DestFileName: TFileName; CompressionLevel: integer=2); deprecated;

Backup of the opened Database into a .gz compressed file
- DEPRECATED: use DB.BackupBackground() instead
- this method doesn't use the SQLite Online Backup API, but low-level database file copy which may lock the database process if the data is consistent - consider using DB.BackupBackground() method instead
- database is closed, VACCUUMed, compressed into .gz file, then reopened
- default compression level is 2, which is very fast, and good enough for a database file content: you may change it into the default 6 level

function ComputeDBStats: variant; overload;

Retrieves the per-statement detailed timing, as a TDocVariantData
### function Restore(const ContentToRestore: RawByteString): boolean;

*Restore a database content on the fly*
- database is closed, source DB file is replaced by the supplied content, then reopened
- there are cases where this method will fail and return FALSE: consider shutting down the server, replace the file, then relaunch the server instead

### function RestoreGZ(const BackupFileName: TFileName): boolean;

*Restore a database content on the fly, from a .gz compressed file*
- database is closed, source DB file is replaced by the supplied content, then reopened
- there are cases where this method will fail and return FALSE: consider shutting down the server, replace the file, then relaunch the server instead

### function RetrieveBlobFields(Value: TSQLRecord): boolean; override;

*Overridden method for direct SQLite3 database engine call*
- it will retrieve all BLOB fields at once, in one SQL statement

### function StoredProcExecute(const aSQL: RawUTF8; StoredProc: TOnSQLStoredProc): boolean;

*Execute one SQL statement, and apply an Event to every record*
- lock the database during the run
- call a fast "stored procedure"-like method for each row of the request; this method must use low-level DB access in any attempt to modify the database (e.g. a prepared TSQLRequest with Reset+Bind+Step), and not the TSQLRestServerDB.Engine*() methods which include a Lock(); this Lock() is performed by the main loop in EngineExecute() and any attempt to such high-level call will fail into an endless loop
- caller may use a transaction in order to speed up StoredProc() writing
- intercept any DB exception and return false on error, true on success

### function TableMaxID(Table: TSQLRecordClass): TID; override;

*Search for the last inserted ID in a table*
- will execute not default select max(rowid) from Table, but faster
  select rowid from Table order by rowid desc limit 1

### function TransactionBegin(aTable: TSQLRecordClass; SessionID: cardinal=1): boolean; override;

*Begin a transaction (implements REST BEGIN Member)*
- to be used to speed up some SQL statements like Insert/Update/Delete
- must be ended with Commit on success
- must be aborted with Rollback if any SQL statement failed
- return true if no transaction is active, false otherwise

### function UpdateBlobFields(Value: TSQLRecord): boolean; override;

*Overridden method for direct SQLite3 database engine call*
- it will update all BLOB fields at once, in one SQL statement

### procedure AdministrationExecute(const DatabaseName, SQL: RawUTF8; var result: TServiceCustomAnswer); override;

*Used e.g. by IAdministratedDaemon to implement "pseudo-SQL" commands*

### procedure Commit(SessionID: cardinal=1; RaiseException: boolean=false); override;

*End a transaction (implements REST END Member)*
- write all pending SQL statements to the disk
procedure ComputeDBStats(out result: variant); overload;
Retrieves the per-statement detailed timing, as a TDocVariantData

procedure CreateMissingTables(user_version: cardinal=0; Options: TSQLInitializeTableOptions=[]); override;
Missing tables are created if they don’t exist yet for every TSQLRecord class of the Database Model
- you must call explicitly this before having called StaticDataCreate()
- all table description (even Unique feature) is retrieved from the Model
- this method also create additional fields, if the TSQLRecord definition has been modified; only field adding is available, field renaming or field deleting are not allowed in the Framework (in such cases, you must create a new TSQLRecord type)

procedure DefinitionTo(Definition: TSynConnectionDefinition); override;
Save the TSQLRestServerDB properties into a persistent storage object
- RegisteredClassCreateFrom() will expect Definition.DatabaseName to store the DBFileName, and optionally encrypt the file using Definition.Password

procedure FlushInternalDBCache; override;
Call this method when the internal DB content is known to be invalid
- by default, all REST/CRUD requests and direct SQL statements are scanned and identified as potentially able to change the internal SQL/JSON cache used at SQLite3 database level; but some virtual tables (e.g. TSQLRestStorageExternal classes defined in SQLite3DB) could flush the database content without proper notification
- this overridden implementation will call TSQLDataBase.CacheFlush method

procedure FlushStatementCache;
Call this method to flush the internal SQL prepared statements cache
- you should not have to flush the cache, only e.g. before a DROP TABLE
- in all cases, running this method would never harm, nor be slow

procedure InitializeEngine; virtual;
Initialize the associated DB connection
- called by Create and on Backup/Restore just after DB.DBOpen
- will register all *_in() functions for available TSQLRecordRTree
- will register all modules for available TSQLRecordVirtualTable*ID with already registered modules via RegisterVirtualTableModule()
- you can override this method to call e.g. DB.RegisterSQLFunction()

procedure RollBack(SessionID: cardinal=1); override;
Abort a transaction (implements REST ABORT Member)
- restore the previous state of the database, before the call to TransactionBegin

property DB: TSQLDataBase read fDB;
Associated database

property StatementLastException: RawUTF8 read fStatementLastException;
Contains some textual information about the latest Exception raised during SQL statement execution

property StatementPreparedSelectQueryPlan: boolean read fStatementPreparedSelectQueryPlan write fStatementPreparedSelectQueryPlan;
Executes (therefore log) the QUERY PLAN for each prepared statement
property StatementTruncateSQLLogLen: integer read fStatementTruncateSQLLogLen write fStatementTruncateSQLLogLen;

After how many bytes a sl|SQL statement log entry should be truncated
- default is 0, meaning no truncation
- typical value is 2048 (2KB), which will avoid any heap allocation

**TSQLRestClientDB** = class(TSQLRestClientURI)

REST client with direct access to a SQLite3 database
- a hidden TSQLRestServerDB server is created and called internally

Used for DI-2.2.1 (page 2548).

constructor Create(aRunningServer: TSQLRestServerDB); reintroduce; overload;

Initialize the class, for an existing TSQLRestServerDB
- the client TSQLModel will be cloned from the server's one
- the TSQLRestServerDB and TSQLDatabase instances won't be managed by the client, but will access directly to the server

constructor Create(aClientModel, aServerModel: TSQLModel; const aDBFileName: TFileName; aServerClass: TSQLRestServerDBClass; aHandleUserAuthentication: boolean=false; const aPassword: RawUTF8=''; aDefaultCacheSize: integer=10000); reintroduce; overload;

Same as above, from a SQLite3 filename specified
- an internal TSQLDataBase will be created internally and freed on Destroy
- aServerClass could be TSQLRestServerDB by default
- if specified, the password will be used to cypher this file on disk (the main SQLite3 database file is encrypted, not the wal file during run)

constructor Create(aClientModel, aServerModel: TSQLModel; aDB: TSQLDataBase; aServerClass: TSQLRestServerDBClass; aHandleUserAuthentication: boolean=false); reintroduce; overload;

Initializes the class, and creates an internal TSQLRestServerDB to internally answer to the REST queries
- aServerClass could be TSQLRestServerDB by default

destructor Destroy; override;

Release the server

function List(const Tables: array of TSQLRecordClass; const SQLSelect: RawUTF8='ID'; const SQLWhere: RawUTF8=''): TSQLTableJSON; override;

Retrieve a list of members as a TSQLTable (implements REST GET Collection)
- this overridden method call directly the database to get its result, without any URI() call, but with use of DB JSON cache if available
- other TSQLRestClientDB methods use URI() function and JSON conversion of only one record properties values, which is very fast

property DB: TSQLDataBase read getDB;

Associated database

property Server: TSQLRestServerDB read fServer;

Associated Server
**Define a Virtual Table module for a stand-alone SQLite3 engine**
- it’s not needed to free this instance: it will be destroyed by the SQLite3 engine together with the DB connection

**Function** `FileName(const aTableName: RawUTF8): TFName; override;`
*Retrieve the file name to be used for a specific Virtual Table*
- overridden method returning a file located in the DB file folder, and " if the main DB was created as SQLite3_MEMORY_DATABASE_NAME (i.e. 'memory:' so that no file should be written)
- of course, if a custom FilePath property value is specified, it will be used, even if the DB is created as SQLite3_MEMORY_DATABASE_NAME

**Procedure** `Attach(aDB: TSQLDataBase);`
*Initialize the module for a given DB connection*
- internally set fModule and call sqlite3_create_module_v2(fModule)
- will raise EBusinessLayerException if aDB is incorrect, or SetDB() has already been called for this module
- will call sqlite3_check() to raise the corresponding ESQLite3Exception
- in case of success (no exception), the SQLite3 engine will release the module by itself; but in case of error (an exception is raised), it is up to the caller to intercept it via a try..except and free the TSQLVirtualTableModuleSQLite3 instance

**Property** `DB: TSQLDataBase read fDB;`
*The associated SQLite3 database connection*

**Define a Virtual Table module for a TSQLRestServerDB SQLite3 engine**

**Constructor** `Create(aClass: TSQLVirtualTableClass; aServer: TSQLRestServer); override;`
*Register the Virtual Table to the database connection of a TSQLRestServerDB server*
- in case of an error, an exception will be raised

**REST storage sharded over several SQLite3 instances**
- numerotated '*0000.dbs' SQLite3 files would contain the sharded data
- here *.dbs is used as extension, to avoid any confusion with regular SQLite3 database files (*.db or *.db3)
- when the server is off (e.g. on periodic version upgrade), you may safely delete/archive some oldest *.dbs files, for easy and immediate purge of your database content: such process would be much faster and cleaner than regular "DELETE FROM TABLE WHERE ID < ?" + "VACUUM" commands
constructor Create(aClass: TSQLRecordClass; aServer: TSQLRestServer; aShardRange: TID; aOptions: TSQLRestStorageShardOptions=[]; const aShardRootFileName: TFileName=''; aMaxShardCount: integer=100; aSynchronous: TSQLSynchronousMode=smOff; aCacheSizePrevious: integer=250; aCacheSizeLast: integer=500); reintroduce; virtual;

Initialize the table storage redirection for sharding over SQLite3 DB
- if no aShardRootFileName is set, the executable folder and stored class table name would be used
- typical use may be:
  Server.StaticDataAdd(TSQLRestStorageShardDB.Create(TSQLRecordSharded,Server,500000))
- you may define some low-level tuning of SQLite3 process via aSynchronous / aCacheSizePrevious / aCacheSizeLast / aMaxShardCount parameters, if the default smOff / 1MB / 2MB / 100 values are not enough

property ShardRootFileName: TFileName read fShardRootFileName;

Associated file name for the SQLite3 database files
- contains the folder, and root file name for the storage
- each shard would end with its 4 digits index: actual file name would append '0000.dbs' to this ShardRootFileName

Types implemented in the mORMotSQLite3 unit

TSQLRestServerDBClass = class of TSQLRestServerDB;
Class-reference type (metaclass) of a REST server using SQLite3 as main engine

Functions or procedures implemented in the mORMotSQLite3 unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegisterVirtualTableModule</td>
<td>Initialize a Virtual Table Module for a specified database</td>
<td>2389</td>
</tr>
</tbody>
</table>

function RegisterVirtualTableModule(aModule: TSQLVirtualTableClass; aDatabase: TSQLDataBase): TSQLVirtualTableModule;

Initialize a Virtual Table Module for a specified database
- to be used for low-level access to a virtual module, e.g. with TSQLVirtualTableLog
- when using our ORM, you should call TSQLModel.VirtualTableRegister() instead to associate a TSQLRecordVirtual class to a module
- returns the created TSQLVirtualTableModule instance (which will be a TSQLVirtualTableModuleSQLite3 instance in fact)
- will raise an exception of failure
27.66. mORMotToolBar.pas unit

*Purpose*: ORM-driven Office 2007 Toolbar for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The `mORMotToolBar` unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.3.1.1</td>
<td>Database Grid Display</td>
<td>2549</td>
</tr>
<tr>
<td>DI-2.3.1.2</td>
<td>RTTI generated Toolbars</td>
<td>2549</td>
</tr>
</tbody>
</table>

Units used in the `mORMotToolBar` unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| `mORMot`      | Common ORM and SOA classes for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1899 |
| `mORMotI18n`  | Internationalization (I18n) routines and classes for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2322 |
| `mORMotReport`| Reporting unit  
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2352 |
| `mORMotUI`    | Grid to display database content for mORMot  
- this unit is a part of the freeware mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2404 |
| `mORMotUILogin`| Some common User Interface functions and dialogs for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2418 |
| `SynCommons`  | Common functions used by most Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 717 |
| `SynGdiPlus`  | GDI+ library API access  
- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic  
- make available most useful GDI+ drawing methods  
- allows AntiAliased rendering of any EMF file using GDI+  
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1350 |
| `SynTable`    | Filter/database/cache/buffer/security/search/multithread/OS features  
- as a complement to SynCommons, which tended to increase too much  
- licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1721 |
Synopse mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynTaskDialog</td>
<td>Implement TaskDialog window (native on Vista/Seven, emulated on XP)</td>
<td>1824</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynZip</td>
<td>Low-level access to ZLib compression (1.2.5 engine version)</td>
<td>1845</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

**mORMotToolBar class hierarchy**

**Objects implemented in the mORMotToolBar unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFreeShortcut</td>
<td>A simple object to get one char shortcuts from caption value</td>
<td>2392</td>
</tr>
<tr>
<td>TSQLCustomToolBar</td>
<td>Create one or more toolbars in a ribbon page, according to an enumeration of actions</td>
<td>2396</td>
</tr>
<tr>
<td>TSQLLister</td>
<td>A hidden component, used for handling toolbar buttons of actions to be performed on a TSQLRecordClass list</td>
<td>2394</td>
</tr>
<tr>
<td>TSQLRibbon</td>
<td>Store some variables common to all pages, i.e. for the whole ribbon</td>
<td>2398</td>
</tr>
<tr>
<td>TSQLRibbonTab</td>
<td>Store the UI elements and data, one per each Table</td>
<td>2396</td>
</tr>
<tr>
<td>TSynPage</td>
<td>A Ribbon page, which will contain some toolbars for a TSQLRecord class</td>
<td>2392</td>
</tr>
<tr>
<td>TSynPager</td>
<td>The ribbon pager, which will contain one page per TSQLRecord class</td>
<td>2393</td>
</tr>
</tbody>
</table>
Objects | Description | Page
---|---|---
TSynToolBar | A toolbar on a Ribbon page | 2392
TSynToolButton | A button on the Ribbon toolbars, corresponding to one action | 2392

TFreeShortCut = object(TObject)

A simple object to get one char shortcuts from caption value

Values: TFreeShortCutSet;

Bit set for already used short cut, from 'A' to 'Z'

function FindFreeShortCut(const aCaption: string): string;

Attempt to create free shortcut of one char length, from a Caption: try every character of aCaption, from left to right
- returns " if no shortcut calculation was possible

TSynToolBarButton = class(TToolBar)

A button on the Ribbon toolbars, corresponding to one action

constructor Create(aOwner: TComponent); override;

This class will have AutoSize set to true

function Images: TCustomImageList;

The associated image list, i.e. TToolBar(Owner).Images

procedure DoDropDown;

Display drop down menu

TSynToolBar = class(TToolBar)

A toolbar on a Ribbon page

function CreateToolButton(ButtonClick: TNotifyEvent; iAction, ImageListFirstIndex: integer; const ActionName, ActionHints: string; var ShortCutUsed: TFreeShortCut; ButtonWidth: integer; Images: TCustomImageList): TSynToolButton;

Create a button on the toolbar

TSynPage = class(TTabSheet)

A Ribbon page, which will contain some toolbars for a TSQLRecord class

function CreateToolBar(AddToList: boolean=true): TSynToolBar;

Add a TSynToolBar to the page list
- then call TSynToolBar.CreateToolButton to add some buttons

procedure ToolBarCreated;

Call this event when all toolbars have been created
- it will create the captions under the toolbars
- can be call multiple times, when a toolbar has been added and filled will all its buttons
property ToolBarCount: integer read GetToolBarCount;
  Number of TSynToolBar in the page list

property ToolBars[aIndex: integer]: TSynToolBar read GetToolBar;
  Access to the TSynToolBar list of this page

TSynPager = class(TPageControl)
  The ribbon pager, which will contain one page per TSQLRecord class

function AddPage(const aCaption: string): integer; overload;
  Create a new page with the specified caption

function AddPage(aPage: TSynPage): integer; overload;
  Add a page instance

class function CreatePager(aOwner: TCustomForm; NoTabVisible: boolean=false): TSynPager;
  Create the ribbon pager on a form
  - reserve some above space for groups, caption and min/max/close buttons, so that
  FormNoCaption method can be called later

function TabGroupsAdd(TabIndexStart, TabIndexEnd: integer; const aCaption: string): TLabel;
  Create a group label starting for the given page indexes

procedure FormNoCaption;
  Hide TSynForm caption bar and put caption and buttons at groups right

property ActivePageIndex: integer read GetActivePageIndex write SetActivePageIndex;
  Force OnChange event to be triggered

property Caption: TLabel read GetCaption;
  The label on TopMostPanel, i.e. the TSynForm(Owner).NoCaption

property HelpButton: TSynToolButton read GetHelpButton;
  The help button to be available on the ribbon

property OnDblClick;
  Publish this property, e.g. to close a tab by a double click

property Pages[aIndex: Integer]: TSynPage read GetSynPage;
  Mimic TTabSheet.Pages property

property TopMostPanel: TPanel read fTopMostPanel;
  The panel added above the pager, containing groups, caption and buttons

property TopPanel: TPanel read fTopPanel;
  The panel containing this TSynPager
**TSQLLister = class(TComponent)**

A hidden component, used for handling toolbar buttons of actions to be performed on a TSQLRecordClass list

*Used for DI-2.3.1.1 (page 2549), DI-2.3.1.2 (page 2549).*

```pascal
constructor Create(aOwner: TComponent; aClient: TSQLRestClientURI; aClass: TSQLRecordClass; aGrid: TDrawGrid; aIDColumnHide: boolean; aPager: TSynPager; aImageList32,aImageList16: TImageList; aOnButtonClick: TSQLListerEvent; aOnValueText: TValueTextEvent; const aGridSelect: RawUTF8= ''; aHideDisabledButtons: boolean=false; aHeaderCheckboxSelectsInsteadOfSort: Boolean=false); reintroduce; overload;
```

Initialize the lister for a specified Client and Class
- the possible actions are retrieved from the Client TSQLModel
- a single page is used for a list of records, specified by their unique class
- a single page can share multiple toolbars
- both TImageList will be used to display some images in Action buttons (32 pixels wide) and Popup Menu (16 pixels wide)
- if aGrid has no associated TSQLTableToGrid, a default one is created retrieving a list of records with aGridSelect about the aClass Table from aClient, with the ID column hidden (no TSQLTableToGrid will be created if aGridSelect is '')
- aOnButtonClick is called with a specified action if a button is clicked, or with ActionValue=0 each time a row is selected

*Used for DI-2.3.1.1 (page 2549).*

```pascal
constructor Create(aOwner: TComponent; aClient: TSQLRestClientURI; aClass: TSQLRecordClass; aGrid: TDrawGrid; aIDColumnHide: boolean; aPager: TSynPager; aImageList32,aImageList16: TImageList; aOnButtonClick: TSQLListerEvent; aOnValueText: TValueTextEvent; aTable: TSQLTable; aHideDisabledButtons, aHeaderCheckboxSelectsInsteadOfSort: boolean); reintroduce; overload;
```

Same as above, but with a specified TSQLTable

*Used for DI-2.3.1.1 (page 2549).*

```pascal
function ActionHint(const Action): string;
```

Retrieve a ready to be displayed hint for a specified action
- returns the Hint caption of the corresponding button, or '' if not existing

```pascal
class function AddPage(aOwner: TSynPager; aClass: TSQLRecordClass; const
CustomCaption: string; CustomCaptionTranslate: boolean): TSynPage;
```

Add a page (if not already) for a corresponding TSQLRecordClass
- the TSynPage tag property will contain integer(aClass)
- the TSynPage caption is expanded and translated from aClass with
  LoadResStringTranslate(aClass.SQLTableName) or taken directly from CustomCaption if a value is specified (with translation if CustomCaptionTranslate is set)

```pascal
function FindButton(ActionIndex: integer): TSynToolButton;
```

Find associate Button for an action

```pascal
function FindMenuItem(ActionIndex: integer): TMenuItem;
```

Find associate popup Menu item for an action
class function FindPage(aOwner: TSynPager; aClass: TSQLRecordClass): integer;
  Retrieve the page index from a TSQLRecordClass
  - the TSynPage tag property contains integer(aClass)

function NewMenuItem(Menu: TPopupMenu; const aCaption: string; ImageIndex: integer=-1; SubMenu: TMenuItem=nil; OnClick: TNotifyEvent=nil; itemEnabled: boolean=true): TMenuItem;
  Create a menu item, and add it to a menu

function SetToolBar(const aToolBarName: string; const aActions; ActionIsNotButton: pointer): TSynToolBar;
  Add or update a ToolBar with a specific actions set
  - a single page can share multiple toolbars, which caption name must be identical between calls for genuine buttons
  - if the ToolBar is already existing, the status of its Action buttons is enabled or disabled according to the actions set
  - aActions must point to a set of enumerates, as defined by Client.Model.SetActions(TypeInfo(..))
  - first call once this procedure to create the toolbar buttons, then call it again to update the enable/disable status of the buttons

procedure CreateSubMenuItem(const aCaption: string; ActionIndex: integer; OnClick: TNotifyEvent; ImageIndex: integer=-1; Tag: integer=0; itemEnabled: boolean=true);
  Create a sub menu item to both button and menu item for an action
  - if aCaption is '', erase any previous menu

procedure OnDrawCellBackground(Sender: TObject; ACol, ARow: Longint; Rect: TRect; State: TGridDrawState);
  Can be used by any TSQLTableToGrid
  - to draw marked rows with a highlighted color
  - with respect to the Toolbar theming

property ActionHints: string read fActionHints write fActionHints;
  The Hints captions to be displayed on the screen
  - must be set before SetToolBar() method call
  - one action (starting with actMark) each line

property Client: TSQLRestClient read fClient;
  The associated Client

property Grid: TDrawGrid read fGrid;
  The associated Grid display

property ImageList16: TImageList read fImageList16;
  TImageList used to display some images in Action buttons

property ImageList32: TImageList read fImageList32;
  TImageList used to display some images in Action buttons

property Menu: TSynPopupMenu read fMenu;
  The Popup Menu, displayed with the Grid

property OnMarkAction: TMarkActionEvent read fOnMarkAction write fOnMarkAction;
  A callback event, triggered after actMark*/actUnmarkAll has been executed
property Page: TSynPage read fPage;
  The associated Page on the Office 2007 menu

property RecordClass: TSQLRecordClass read fClass;
  The associated record class

property ReportDetailedIndex: integer read fReportDetailedIndex;
  Set to a "Details" level, according to the bsCheck button pushed
  - set to the Action index which is currently available

property TableToGrid: TSQLTableToGrid read fTableToGrid;
  The associated TSQLTableToGrid hidden component

TSQLCustomToolBar = object(TObject)

Create one or more toolbars in a ribbon page, according to an enumeration of actions
- use a similar layout and logic as TSQLLister.SetToolBar() method above
- to be used for custom forms (e.g. preview or edit) or to add some custom buttons to a previously
  created one by TSQLLister.SetToolBar()
- simply set the associated objects via the Init() method, then call AddToolBar() for every toolbar
  which need to be created

Used for DI-2.3.1.2 (page 2549).

function AddToolBar(const ToolBarName: string; ActionsBits: pointer=nil; ButtonWidth: integer=60): TSynToolBar;
  Call this method for every toolbar, with appropriate bits set for its buttons

function CreateSubMenu(pButtonItem: integer; const aCaption: string; aOnClick: TNotifyEvent; aTag: integer=0): TMenuItem;
  Create a popup menu item for a button
  - call with aCaption void to clear the menu first
  - then call it for every menu entry

procedure Init(aToolbarOrPage: TControl; aEnum: PTypeInfo; aButtonClick: TNotifyEvent; aImageList: TImageList; const aActionHints: string;
aImageListFirstIndex: integer=0);
  Call this method first to initialize the ribbon
  - if aToolbarOrPage is a TCustomForm, this form will become a
  - if aToolbarOrPage is a TSynPager descendant, a new page is created and added to this
    TSynPager, and used for toolbars adding
  - if aToolbarOrPage is a TSynPage descendant, the toolbar is added to this specified Page

Used for DI-2.3.1.2 (page 2549).

TSQLRibbonTab = class(TObject)

Store the UI elements and data, one per each Table

Used for DI-2.3.1.2 (page 2549).

CurrentRecord: TSQLRecord;
  A current record value
FrameLeft: TFrame;
The frame containing associated the List, left side of the Page

FrameRight: TFrame;
The frame containing associated Details, right side to the list

FrameSplit: TSplitter;
Allows List resizing

List: TDrawGrid;
Associated table list

Lister: TSQLLister;
To associate Class, Actions, Ribbon and Toolbars

Page: TSynBodyPage;
Associate Client Body Page

Parameters: PSQLRibbonTabParameters;
Associated Tab settings used to create this Ribbon Tab

Report: TGDIPages;
The associated Report, to display the page
- exists if aTabParameters.Layout is not IIClient, and if aTabParameters.NoReport is false

Tab: TSynPage;
Associate Tab in the Ribbon

Table: TSQLRecordClass;
Associated TSQLRecord

TableIndex: integer;
Associated TSQLRecord index in database Model

TableToGrid: TSQLTableToGrid;
To provide the List with data from Client

ViewToolBar: TSynToolBar;
The "View" toolbar on the associated Ribbon Tab

constructor Create(ToolBar: TSynPager; Body: TSynBodyPager; aImageList32, aImageList16: TImageList; var aPagesShortCuts: TFreeShortCut; const aTabParameters: TSQLRibbonTabParameters; Client: TSQLRestClientURI; aUserRights: TSQLFieldBits; aOnValueText: TValueTextEvent; SetAction: TSQLRibbonSetActionEvent; const ActionsTCPicationCSV, ActionsHintCaption: string; ActionIsNotButton: pointer; aOnActionClick: TSQLListerEvent; ViewToolbarIndex: integer; aHideDisabledButtons, aHeaderCheckboxSelectsInsteadOfSort: boolean);
Create all the UI elements for a specific Table/Class
- create a new page for this Table/Class
- populate this page with available Toolbars
- populate all Toolbars with action Buttons

destructor Destroy; override;
Release associated memory
function AskForAction(const ActionCaption, aTitle: string; Client: TSQLRest; DontAskIfOneRow, ReturnMarkedIfSomeMarked: boolean): integer;
  Ask the User where to perform an Action
  - return 100 if "Apply to Selected" was choosen
  - return 101 if "Apply to Marked" was choosen
  - return any other value if Cancel or No was choosen

function Retrieve(Client: TSQLRestClient; ARow: integer; ForUpdate: boolean=false): boolean;
  Retrieve CurrentRecord from server

procedure AddReportPopupMenuOptions(Menu: TPopupMenu; OnClick: TNotifyEvent);
  Add the report options to the specified menu

procedure CustomReportPopupMenu(OnClick: TNotifyEvent; ParamsEnum: PTypeInfo; ParamsEnabled: pointer; const Values: array of PBoolean);
  Used to customize the popup menu of the associated Report
  - this method expect two standard handlers, and a custom enumeration type together with its
    (bit-oriented) values for the current Ribbon Tab
  - caller must supply an array of boolean pointers to reflect the checked state of every popup
    menu item entry

procedure ReportClick(Sender: TObject);
  Triggered when a report popup menu item is clicked

property CurrentID: TID read GetCurrentID;
  Retrieve the current selected ID of the grid
  - returns 0 if no row is selected

property ReportPopupParamsEnabled: pointer read FReportPopupParamsEnabled;
  Pointer to the set of available popup menu parameters for this report

property ReportPopupValues: TPBooleanDynArray read FReportPopupValues;
  Pointers to every popup menu items data

TSQLRibbon = class(TObject)
  Store some variables common to all pages, i.e. for the whole ribbon
  Used for DI-2.3.1.2 (page 2549).

Page: array of TSQLRibbonTab;
  The pages array

ShortCuts: TFreeShortCut;
  Store the keyboard shortcuts for the whole ribbon
**constructor** Create(Owner: TCustomForm; ToolBar: TSynPager; Body: TSynBodyPager; aImageList32,aImageList16: TImageList; Client: TSQLRestClientURI; aUserRights: TSQLFieldBits; aOnValueText: TValueTextEvent; SetAction: TSQLRibbonSetActionEvent; const ActionsTBCaptionCSV, ActionsHintCaption: string; ActionIsNotButton: pointer; aOnActionClick: TSQLListerEvent; RefreshActionIndex, ViewToolbarIndex: integer; aHideDisabledButtons: boolean; PagesCount: integer; TabParameters: PSQLRibbonTabParameters; TabParametersSize: integer; const GroupCSV: string; const BackgroundPictureResourceNameCSV: string=''; aHeaderCheckboxSelectsInsteadOfSort: boolean=false); reintroduce; virtual;

*Initialize the Pages properties for this ribbon*
- this constructor must be called in the Owner.OnCreate handler (not in OnShow)
- most parameters are sent back to the SQLRibbonTab.Create constructor
- if BackgroundPictureResourceNameCSV is set, the corresponding background pictures will be extracted from resources and displayed behind the ribbon toolbar, according to the group

*Used for DI-2.3.1.2 (page 2549).*

**destructor** Destroy; override;
*Release associated memory*

**function** AddToReport(aReport: TGDPages; aRecord: TSQLRecord; WithEvents: Boolean; CSVFieldNames: PUTF8Char=nil; CSVFieldNameToHide: PUTF8Char=nil; OnCaptionName: TOnCaptionName=nil; ColWidthName: Integer=40; ColWidthValue: integer=60): string; overload;

*Add the specified fields content to the report*
- by default, all main fields are displayed, but caller can specify custom field names as Comma-Separated-Values
- retrieve the main Caption of this record (e.g. the "Name" field value)

**function** DeleteMarkedEntries(aTable: TSQLRecordClass; const ActionHint: string): Boolean;
*Generic method which delete either the current selected entry, either all marked entries*
- returns TRUE if deletion was successful, or FALSE if any error occured

**function** ExportRecord(aTable: TSQLRecordClass; aID: TID; const ActionHint: string; OpenAfterCreation: boolean=true): TFileName;
*Generic method which export the supplied record*
- display the save dialog before
- only two formats are available here: Acrobat (.pdf) and plain text (.txt)
- returns the exported file name if export was successful, or " if any error occured
- by default, the report is created by using the CreateReport method

**function** FindButton(aTable: TSQLRecordClass; aActionIndex: integer): TSynToolButton;
*Retrieve the reference of a given button of the ribbon*
- useful to customize the Ribbon layout, if automatic generation from RTTI don't fit exactly your needs, or even worse marketing's know-how ;)
- called by SetButtonHint method

**function** GetActivePage: TSQLRibbonTab;
*Retrieve the current TSQLRibbonTab instance on the screen*
- returns nil if no page is currently selected
function GetPage(aRecordClass: TSQLRecordClass): integer;
  Retrieve the index of a given Pages[]
  - returns -1 if this page was not found

function GetParameter(aPageIndex: Integer): PSQLRibbonTabParameters; overload;
  Retrieve the TSQLRibbonTabParameters associated to a Ribbon tab, from its index
  - returns nil if the specified page index is not valid

function GetParameter(aTable: TSQLRecordClass): PSQLRibbonTabParameters; overload;
  Get the TSQLRibbonTabParameters associated to a Ribbon tab, from its table
  - returns nil if the specified table is not valid

function MarkedEntriesToReport(aTable: TSQLRecordClass; const ColWidths: array of integer; aRep: TGDIPages=nil): TGDIPages;
  Generic method which print the all marked entries of the supplied table

function RefreshClickHandled(Sender: TObject; RecordClass: TSQLRecordClass; ActionValue: integer; out Tab: TSQLRibbonTab): boolean;
  Handle a ribbon button press
  - returns TRUE if a Refresh command has been processed (caller should exit) and a refresh timer command has been set
  - returns FALSE if the caller must handle the action

procedure AddToReport(aReport: TGDIPages; Table: TSQLTable; const ColWidths: array of integer); overload;
  Add the specified database Table Content to the report
  - if ColWidths are not specified (that is set to []), their values are caculated from the Table content columns

procedure BodyResize(Sender: TObject);
  Resize the lists according to the body size

procedure CreateReport(aPageIndex: Integer); overload;
  Create a report for the specified page index
  - the report must be created from the Page[aPageIndex].CurrentRecord record content
  - call the CreateReport virtual method

procedure CreateReport(aTable: TSQLRecordClass; aID: TID; aReport: TGDIPages; AlreadyBegan: boolean=false); overload; virtual;
  Create a report for the specified page index
  - this default method create a report with the content of all fields, except those listed in the corresponding TSQLRibbonTabParameters.EditFieldNameToHideCSV value

procedure GotoRecord(aRecord: TSQLRecord; ActionToPerform: integer=0); overload;
  Make a specified record available to the UI
  - select tab and record index
  - if ActionToPerform is set, the corresponding action is launched

procedure GotoRecord(aTable: TSQLRecordClass; aID: TID; ActionToPerform: integer=0); overload;
  Make a specified record available to the UI
  - select tab and record index
  - if ActionToPerform is set, the corresponding action is launched
procedure Refresh(aTable: TSQLRecordClass=nil);

Refresh the specified page content
- by default, refresh the current page content
- calls internally RefreshClickHandled method

procedure SetButtonHint(aTable: TSQLRecordClass; aActionIndex: integer; const aHint: string);

Customize the Hint property of any button
- will test the button is available (avoid any GPF error)

procedure ToolBarChange(Sender: TObject);

Trigger this event when a page changed on screen
- will free GDI resources and unneeded memory

procedure WMRefreshTimer(var Msg: TWMTimer);

Must be called by the main form to handle any WM_TIMER message
- will refresh the screen as necessary

property Body: TSynBodyPager read fBody;
The main Pager component used to display the main data (i.e. records list and report) on the Form

property Client: TSQLRestClientURI read fClient write fClient;
The associated Client connection

property Form: TCustomForm read fForm;
The associated Form on screen

property ReportAutoFocus: boolean read fReportAutoFocus write fReportAutoFocus;
If set to TRUE, the right-sided report is focused instead of the left-sided records list

property ToolBar: TSynPager read fToolBar;
The Toolbar component used to display the Ribbon on the Form

Types implemented in the mORMotToolBar unit

TFreeShortCutSet = set of ord('A')..ord('Z');
Used to mark which shortcut keys have already been affected

TMarkActionEvent = procedure(Sender: TObject; RecordClass: TSQLRecordClass;
MarkAction: TSQLAction) of object;
This event is called after actMark*/actUnmarkAll has been executed

TOnCaptionName = function(const Action: RawUTF8; Obj: TObject=nil; Index: integer=-1): string of object;
Event used to customize screen text of property names

TPBooleanDynArray = array of PBoolean;
Used to store the options status

TSQLListerEvent = procedure(Sender: TObject; RecordClass: TSQLRecordClass;
ActionValue: integer) of object;
This event is called when a button is pressed
- here ActionValue contains the ordinal value of the custom button

TSQLRibbonSetActionEvent = function(TabIndex, ToolbarIndex: integer; TestEnabled:
This event provide the action values for a specified toolbar
- first call is to test the action presence, with TestEnabled=false
- a special call is made with ToolBarIndex=-1, in which A should be filled with the marking actions
- second call is to test the action enable/disable state, with TestEnabled=true
- in all cases, should return any customized toolbar caption name, or "

Functions or procedures implemented in the mORMotToolBar unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddIconToImageList</td>
<td>Add an Icon to the supplied TImageList</td>
<td>2402</td>
</tr>
<tr>
<td>CaptionName</td>
<td>Retrieve the ready to be displayed text of the given property</td>
<td>2402</td>
</tr>
<tr>
<td>CreateReportWithIcons</td>
<td>Create a report containing all icons for a given action enumeration</td>
<td>2403</td>
</tr>
<tr>
<td>ImageListStretch</td>
<td>Fill a TImageList from the content of another TImageList</td>
<td>2403</td>
</tr>
<tr>
<td>LoadBitmapFromResource</td>
<td>Load a bitmap from a .png/.jpg file embedded as a resource to the executable</td>
<td>2403</td>
</tr>
<tr>
<td>LoadImageListFromBitmap</td>
<td>Load TImageList bitmaps from a TBitmap</td>
<td>2403</td>
</tr>
<tr>
<td>LoadImageListFromEmbeddedZip</td>
<td>Load TImageList bitmaps from an .zip archive embedded as a ZIP resource</td>
<td>2403</td>
</tr>
<tr>
<td>NewDrawCellBackgroundColor</td>
<td>Draw the cell of a TDrawGrid according to the current Theming of TabAppearance</td>
<td>2403</td>
</tr>
</tbody>
</table>

function AddIconToImageList(ImgList: TCustomImageList; Icon: HIcon): integer;

Add an Icon to the supplied TImageList
- return the newly created index in the image list
- the HIcon handle is destroyed before returning

function CaptionName(OnCaptionName: TOnCaptionName; const Action: RawUTF8; Obj: TObject=nil; Index: integer=-1): string;

Retrieve the ready to be displayed text of the given property
procedure CreateReportWithIcons(ParamsEnum: PTypeInfo; ImgList: TImageList; const Title, Hints: string; StartIndexAt: integer);

Create a report containing all icons for a given action enumeration
- useful e.g. for marketing or User Interface review purposes

procedure ImageListStretch(ImgListSource, ImgListDest: TImageList; BkColor: TColor=clSilver);

Fill a TImageList from the content of another TImageList
- stretching use GDI+ so is smooth enough for popup menu display

function LoadBitmapFromResource(const ResName: string; Instance: THandle=0): TBitmap;

Load a bitmap from a .png/.jpg file embedded as a resource to the executable
- you can specify a library (dll) resource instance handle, if needed

procedure LoadImageListFromBitmap(ImgList: TCustomImageList; Bmp: TBitmap);

Load TImageList bitmaps from a TBitmap
- warning Bmp content can be modified: it could be converted from multi-line (e.g. IDE export format) into one-line (as expected by TImageList.AddMasked)

procedure LoadImageListFromEmbeddedZip(ImgList: TCustomImageList; const ZipName: TFileName);

Load TImageList bitmaps from an .zip archive embedded as a ZIP resource

procedure NewDrawCellBackground(Sender: TObject; ACol, ARow: Integer; Rect: TRect; State: TGridDrawState; Marked: boolean);

Draw the cell of a TDrawGrid according to the current Theming of TabAppearance
27.67. mORMotUI.pas unit

*Purpose*: Grid to display database content for mORMot
- this unit is a part of the freeware mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

The *mORMotUI* unit is quoted in the following items

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.3.1.1</td>
<td>Database Grid Display</td>
<td>2549</td>
</tr>
</tbody>
</table>

Units used in the *mORMotUI* unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>mORMot</em></td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><em>SynCommons</em></td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

```
mORMotUI class hierarchy

TObject TUIComponentsPersist
TLabeledEdit TSynLabeledEdit
THintWindow THintWindowDelayed
TForm TVistaForm
TComponent TSQLTableToGrid
Exception ESynLabeledEdit
```

Objects implemented in the *mORMotUI* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESynLabeledEdit</td>
<td>Exception class raised by TSynIntegerLabeledEdit</td>
<td>2410</td>
</tr>
<tr>
<td>THintWindowDelayed</td>
<td>A THintWindow descendant, with an internal delay to auto-hide</td>
<td>2405</td>
</tr>
<tr>
<td>TSQLTableToGrid</td>
<td>A hidden component, used for displaying a TSQLTable in a TDrawGrid</td>
<td>2406</td>
</tr>
<tr>
<td>TSynLabeledEdit</td>
<td>TLabeledEdit with optional boundaries check of a Variant value</td>
<td>2410</td>
</tr>
</tbody>
</table>
Objects | Description | Page
---|---|---
TUIComponentsPersist | Allow to track and load/save UI components as JSON | 2411
TVistaForm | Vista-enabled TForm descendant | 2411

**THintWindowDelayed = class(THintWindow)**

A THintWindow descendant, with an internal delay to auto-hide
- this component can be used directly with the hint text to be displayed (companion to the controls Hint properties and Application.ShowHint)
- you can specify a time interval for the popup window to be hidden
- this component expects UTF-8 encoded text, and displays it as Unicode

**constructor** Create(aOwner: TComponent); **override**;

Initializes the component

**destructor** Destroy; **override**;

Releases component resources and memory

**function** CalcHintRect(MaxWidth: Integer; **const** AHint: RawUTF8; AData: Pointer): TRect; **reintroduce**;

Overridden method, Unicode ready

**procedure** ShowDelayedString(**const** Text: string; X,Y,Time: integer; FontColor: TColor; AlignLeft: boolean=false); **overload**;

Displays the appropriate Hint Text at a specified screen position
- if string is AnsiString (i.e. for Delphi 2 to 2007), Text is decoded into Unicode (using the current i18n code page) before display
- Time is the maximum text display delay, in milliseconds

**procedure** ShowDelayedString(**const** Text: string; Origin: TControl; X,Y,Time: integer; FontColor: TColor; AlignLeft: boolean=false); **overload**;

Displays the appropriate Hint Text at a position relative to a control
- Text is decoded from Ansi to Unicode (using the current i18n code page) before display
- Time is the maximum text display delay, in milliseconds

**procedure** ShowDelayedUTF8(**const** Text: RawUTF8; X,Y,Time: integer; FontColor: TColor; AlignLeft: boolean=false); **overload**;

Displays the appropriate Hint Text at a specified screen position
- Text is decoded from UTF-8 to Unicode before display
- Time is the maximum text display delay, in milliseconds

**procedure** ShowDelayedUTF8(**const** Text: RawUTF8; Origin: TControl; X,Y,Time: integer; FontColor: TColor; AlignLeft: boolean=false); **overload**;

Displays the appropriate Hint Text at a position relative to a control
- Text is decoded from UTF-8 to Unicode before display
- Time is the maximum text display delay, in milliseconds

**property** Col: integer **read** fCol;

The column number when the hint is displayed

**property** Row: integer **read** fRow;

The row number when the hint is displayed
### TSQLTableToGrid = class(TComponent)

A hidden component, used for displaying a TSQLTable in a TDrawGrid
- just call `TSQLTableToGrid.Create(Grid, Table)` to initiate the association
- the Table will be released when no longer necessary
- any former association by `TSQLTableToGrid.Create()` will be overridden
- handle unicode, column size, field sort, incremental key lookup, hide ID
- Ctrl + click on a cell to display its full unicode content

**Used for DI-2.3.1.1 (page 2549).**

**constructor Create(aOwner: TDrawGrid; aTable: TSQLTable; aClient: TSQLRestClientURI): reintroduce;**

*Fill a TDrawGrid with the results contained in a TSQLTable*

*Used for DI-2.3.1.1 (page 2549).*

**destructor Destroy; override;**

*Release the hidden object*
- will be called by the parent Grid when it is destroyed
- will be called by any future `TSQLTableToGrid.Create()` association
- free the associated TSQLTable and its memory content
- will reset the Grid overridden events to avoid GPF

**function ExpandRowAsString(Row: integer; Client: TObject): string;**

*Read-only access to a particular row values, as VCL text*
- Model is one TSQLModel instance (used to display TRecordReference)
- returns the text as generic string, ready to be displayed via the VCL after translation, for sftEnumerate, sftTimeLog, sftRecord and all other properties
- uses `OnValueText` property Event if defined by caller

**class function From(Grid: TDrawGrid): TSQLTableToGrid;**

*Retrieve the associated TSQLTableToGrid from a specific TDrawGrid*

**function GetMarkedBits: pointer;**

*Retrieve the Marked[] bits array*

**function Refresh(ForceRefresh: Boolean=false; AutoResizeColumns: Boolean=true): boolean;**

*Force refresh paint of Grid from Table data*
- return true if Table data has been successfully retrieved from Client and if data was refreshed because changed since last time
- if ForceRefresh is TRUE, the Client is not used to retrieve the data, which must be already refreshed before this call
- if AutoResizeColumns is TRUE, the column visual width will be re-computed from the actual content - set it to FALSE to avoid it

**function SelectedID: TID;**

*Get the ID of the first selected row, 0 on error (no ID field e.g.)*
- useful even if ID column was hidden with `IDColumnHide`
function SelectedRecordCreate: TSQLRecord;

Retrieve the record content of the first selected row, nil on error
- record type is retrieved via Table.QueryTables[0] (if defined)
- warning: it's up to the caller to Free the created instance after use (you should e.g. embedd the process in a try...finally block):
  Rec := Grid.SelectedRecordCreate;
  if Rec<>nil then
    try
    DoSomethingWith(Rec);
    finally
    Rec.Free;
    end;
- useful even if ID column was hidden with IDColumnHide

procedure AfterRefresh(const aID: TID; AutoResizeColumns: boolean);

Call this procedure after a refresh of the data
- current Row will be set back to aID
- called internal by Refresh function above

procedure DrawCell(Sender: TObject; ACol, ARow: Longint; Rect: TRect; State: TGridDrawState);

Called by the owner TDrawGrid to draw a Cell from the TSQLTable data
- the cell is drawn using direct Win32 Unicode API
- the first row (fixed) is drawn as field name (centered bold text with sorting order displayed with a triangular arrow)

procedure DrawGridColumnDown(Sender: TObject; var Key: Word; Shift: TShiftState);

Called by the owner TDrawGrid when the user presses a key
- used for LEFT/RIGHT ARROW column order change

procedure DrawGridColumnKeyPress(Sender: TObject; var Key: Char);

Called by the owner TDrawGrid when the user presses a key
- used for incremental key lookup

procedure DrawGridColumnMouseDown(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

Called by the owner TDrawGrid when a Cell is clicked by the mouse
- check if the first (fixed) row is clicked: then change sort order
- Ctrl + click to display its full unicode content (see HintText to customize it)

procedure DrawGridColumnMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);

Called by the owner TDrawGrid when the mouse is over a Cell

procedure DrawGridColumnMouseUp(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

Called by the owner TDrawGrid when the mouse is unclicked over a Cell

procedure DrawGridSelectCell(Sender: TObject; ACol, ARow: Integer; var CanSelect: Boolean);

Called by the owner TDrawGrid when a Cell is selected

procedure IDColumnHide;

If the ID column is available, hides it from the grid
procedure OnTableUpdate(State: TOnTableUpdateState);
Used by TSQLRestClientURI.UpdateFromServer() to let the client perform the rows update (for Marked[])

procedure PageChanged;
You can call this method when the list is no more on the screen
- it will hide any pending popup Hint windows, for example

procedure Resize(Sender: TObject);
Call this procedure to automatically resize the TDrawString columns
- can be used as TSQLTableToGrid.From(DrawGrid).Resize();

procedure SetAligned(const aCols: array of cardinal; aAlign: TSQLTableToGridAlign);
Set columns number which must be aligned to non default left layout
- a faster overload to Aligned[] property

procedure SetAlignedByType(aFieldType: TSQLFieldType; aAlign: TSQLTableToGridAlign);
Set column alignment for a given type
- a faster overload to Aligned[] property

procedure SetCustomFormatByType(aFieldType: TSQLFieldType; const aCustomFormat: string);
Set a custom format for all columns of a given type
- a faster overload to CustomFormat[] property
- only support the field types and formats handled by CustomFormat[] property

procedure SetFieldFixedWidth(aColumnWidth: integer);
Force all columns to have a specified width, in pixels

procedure SetFieldLengthMean(const Lengths: RawUTF8; aMarkAllowed: boolean);
Force the mean of characters length for every field
- supply a string with every character value is proportionate to the corresponding column width
- if the character is lowercase, the column is set as centered
- if aMarkAllowed is set, a first checkbox column is added, for reflecting and updating the Marked[] field values e.g.
- if Lengths='', will set some uniform width, left aligned

procedure SetMark(aAction: TSQLAction);
Perform the corresponding Mark/Unmark[All] Action

procedure ShowHintString(const Text: string; ACol, ARow, Time: integer; FontColor: TColor=clBlack);
Display a popup Hint window at a specified Cell position
- expect generic string Text, i.e. UnicodeString for Delphi 2009/2010, ready to be used with the VCL for all Delphi compiler versions

procedure SortChange(ACol: integer);
Toggle the sort order of a specified column

procedure SortForce(ACol: integer; Ascending: boolean; ARow: integer=-1);
Set a specified column for sorting
- if ACol=-1, then the Marked[] rows are shown first, in current sort
### property `Aligned[aCol: cardinal]: TSQLTableToGridAlign read GetAlign write SetAlign;`
- Set individual column alignment

### property `Client: TSQLRestClientURI read fClient;`
- Associated Client used to retrieved the Table data

### property `CurrentFieldOrder: integer read fCurrentFieldOrder;`
- Current field number used for current table sorting

### property `CustomFormat[aCol: cardinal]: string read GetCustomFormat write SetCustomFormat;`
- Set individual column custom format
  - as handled by TSQLTable.ExpandAsString() method, i.e. Format() or FormatFloat()/FormatCurrency() mask for `sftFloat` or `sftCurrency`, or FormatDateTime() mask for `sftDateTime`, `sftDateTimeMS`, `sftTimeLog`, `sftModTime`, `sftCreateTime`, `sftUnixTime`, `sftUnixMSTime`)

### property `DrawGrid: TDrawGrid read GetDrawGrid;`
- Associated `TDrawGrid`
  - just typecast the Owner as `TDrawGrid`

### property `FieldIndexTimeLogForMark: integer read GetFieldIndexTimeLogForMark;`
- Retrieves the index of the `sftTimeLog` first field
  - i.e. the field index which can be used for Marked actions
  - equals -1 if not such field exists

### property `FieldTitleTruncatedNotShownAsHint: boolean read fTruncAsHint write fTruncAsHint;`
- Set to `FALSE` to display the column title as hint when truncated on screen

### property `GridColumnWidths: RawUTF8 read GetGridColumnWidths write SetGridColumnWidths;`
- Retrieve or define the column widths of this grid, as text
  - as a CSV list of the associated DrawGrid.ColWidths[] values

### property `HeaderCheckboxSelectsInsteadOfSort: boolean read fHeaderCheckboxSelectsInsteadOfSort; write fHeaderCheckboxSelectsInsteadOfSort;`
- Set to `TRUE` to let the header check box select/unselect all rows instead of sorting them
  - may be more conventional use of this header check box

### property `Hint: THintWindowDelayed read fHint;`
- Used to display some hint text

### property `MarkAllowed: boolean read fMarkAllowed;`
- True if `Marked[]` is available (add checkboxes at the left side of every row)

### property `MarkAvailable: boolean read GetMarkAvailable;`
- True if any `Marked[]` is checked

### property `Marked[RowIndex: integer]: boolean read GetMarked write SetMarked;`
- Retrieves if a row was previously marked
  - first data row index is 1
property MarkedIsOnlyCurrent: boolean read GetMarkedIsOnlyCurrent;
  True if only one entry is in Marked[], and it is the current one

property MarkedTotalCount: integer read GetMarkedTotalCount;
  Returns the number of item marked or selected
  - if no item is marked, it return 0 even if a row is currently selected

property OnDrawCellBackground: TDrawCellEvent read fOnDrawCellBackground write fOnDrawCellBackground;
  Assign an event here to customize the background drawing of a cell

property OnHintText: THintTextEvent read fOnHintText write fOnHintText;
  Override this event to customize the Ctrl+Mouse click popup text

property OnRightClickCell: TRightClickCellEvent read fOnRightClickCell write fOnRightClickCell;
  Override this event to customize the Mouse right click on a data cell

property OnSelectCell: TSelectCellEvent read fOnSelectCell write fOnSelectCell;
  Override this event to customize the Mouse click on a data cell

property OnSort: TNotifyEvent read fOnSort write fOnSort;
  Override this event to be notified when the content is sorted

property OnValueText: TValueTextEvent read fOnValueText write fOnValueText;
  Override this event to customize the text display in the table

property Table: TSQLTable read fTable;
  Associated TSQLTable to be displayed

ESynLabeledEdit = class(Exception)
  Exception class raised by TSynIntegerLabeledEdit

TSynLabeledEdit = class(TLabeledEdit)
  TLabeledEdit with optional boundaries check of a Variant value

  RaiseExceptionOnError: boolean;
    If true, GetValue() will raise an ESynVariantLabeledEdit exception on any Variant value range error, when the Value property is read

constructor Create(AOwner: TComponent); override;
  Create the component instance

function ToString(NumberOfDigits: integer): string; reintroduce;
  Convert the entered Variant value into a textual representation

function ValidateValue: boolean;
  Return TRUE if the entered value is inside the boundaries

property AdditionalHint: string read FAdditionalHint write FAdditionalHint;
  Some additional popup hint to be displayed
  - by default, the allowed range is displayed: 'Min. Value: #, Max. Value: #'
  - you can specify here some additional text to be displayed when the mouse is hover the component
property Kind: TSynLabeledEditKind read fKind write fKind default sleInteger;
   The kind of value which is currently edited by this TSynLabeledEdit

property MaxValue: Variant read FMaxValue write FMaxValue;
   Highest allowed Variant value

property MinValue: Variant read FMinValue write FMinValue;
   Lowest allowed Variant value

property RangeChecking: boolean read fRangeChecking write fRangeChecking;
   Set to TRUE if MinValue/MaxValue properties must be checked when reading Value property

property Value: Variant read GetValue write SetValue;
   The entered value
   - getting this property will check for in range according to the current MinValue/MaxValue boundaries, if RangeChecking is set
   - if RangeChecking is not set, could return a NULL variant for no data
   - it will sound a beep in case of any out of range
   - it will also raise a ESynVariantLabeledEdit exception if RaiseExceptionOnError is set to TRUE (equals FALSE by default)

TUIComponentsPersist = class(TObject)
   Allow to track and load/save UI components as JSON
   - may be used to persist TEdit / TCheckBox / TComboBox values on a form when the application leaves

function SaveToVariant: variant;
   Save all tracked controls properties as a JSON object

procedure LoadFromFile;
   Fill all tracked controls properties from a local JSON file

procedure LoadFromVariant(const aDoc: variant);
   Fill all tracked controls properties from the supplied JSON object

procedure SaveToFile;
   Save all tracked controls properties as JSON in a local file

procedure TrackControls(const ctrls: array of TComponent);
   Would track .Text and .Checked properties only

property FileName: TFileName read GetFileName write fFileName;
   The local JSON file used for persistence
   - is set to 'executablename.default' if none is specified

TVistaForm = class(TForm)
   Vista-enabled TForm descendant
   - this form will have a button in the TaskBar
   - this form will hide the default Delphi application virtual form
   - this form can be with no caption bar using SetNoCaption method
procedure SetNoCaption(aTopMostPanel: TPanel; aLabelLeft: integer);
   Call this method to hide the Caption bar and replace it with a TPanel

property NoCaptionLabel: TLabel read fNoCaptionLabel;
   The TLabel instance created on NoCaptionPanel to replace the Caption bar

property NoCaptionPanel: TPanel read fNoCaption;
   The TPanel instance replacing the Caption bar

Types implemented in the mORMotUI unit

THintTextEvent = function(Sender: TSQLTable; FieldIndex, RowIndex: Integer; var Text: string): boolean of object;
   Kind of event used to change some text on the fly for popup hint
   - expect generic string Text, i.e. UnicodeString for Delphi 2009/2010, ready to be used with the VCL
   for all Delphi compiler versions

TRightClickCellEvent = procedure(Sender: TSQLTable; ACol, ARow, MouseX, MouseY: Integer) of object;
   Kind of event used to display a menu on a cell right click

TSQLTableToGridAlign = ( alLeft, alCenter, alRight );
   The available alignments of a TSQLTableToGrid cell

TSynLabeledEditKind = ( sleInteger, sleInt64, sleCurrency, sleDouble );
   Diverse kind of values which may be edited by a TSynLabeledEdit

TValueTextEvent = function(Sender: TSQLTable; FieldIndex, RowIndex: Integer; var Text: string): boolean of object;
   Kind of event used to change some text on the fly for grid display
   - expect generic string Text, i.e. UnicodeString for Delphi 2009/2010, ready to be used with the VCL
   for all Delphi compiler versions
   - if the cell at FieldIndex/RowIndex is to have a custom content, shall set the Text variable content
   and return TRUE
   - if returns FALSE, the default content will be displayed

Functions or procedures implemented in the mORMotUI unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddApplicationToFirewall</td>
<td>Allow an application to access the network through the Windows firewall</td>
<td>2413</td>
</tr>
<tr>
<td>AddPortToFirewall</td>
<td>Open a firewall port on the current computer</td>
<td>2413</td>
</tr>
<tr>
<td>ClearTypeEnable</td>
<td>Enable the ClearType font display</td>
<td>2413</td>
</tr>
<tr>
<td>CreateAnIcon</td>
<td>Create an Icon</td>
<td>2413</td>
</tr>
<tr>
<td>DrawCheckBox</td>
<td>Draw a CheckBox in the Canvas Handle of the Wwindow hWnd, in the middle of the Rect coordinates</td>
<td>2413</td>
</tr>
<tr>
<td>FillStringGrid</td>
<td>Fill TStringGrid.Cells[] with the supplied data</td>
<td>2413</td>
</tr>
</tbody>
</table>
### Functions or procedures

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetShellFolderPath</td>
<td>Get the corresponding windows folder, from its ID</td>
<td>2413</td>
</tr>
<tr>
<td>HideAppFormTaskBarButton</td>
<td>Low level VCL routine in order to hide the application from Windows task bar</td>
<td>2414</td>
</tr>
<tr>
<td>IsClearTypeEnabled</td>
<td>Test if the ClearType is enabled for font display</td>
<td>2414</td>
</tr>
<tr>
<td>Register</td>
<td>Register the TSynIntegerLabeledEdit component in the IDE toolbar</td>
<td>2414</td>
</tr>
</tbody>
</table>

**procedure** AddApplicationToFirewall(const EntryName, ApplicationPathAndExe: string);

*Allow an application to access the network through the Windows firewall*
- works on Windows WP, Vista and Seven
- caller process must have the administrator rights (this is the case for a setup program)

**procedure** AddPortToFirewall(const EntryName: string; PortNumber: cardinal);

*Open a firewall port on the current computer*
- works on Windows XP, Vista and Seven
- caller process must have the administrator rights (this is the case for a setup program)

**procedure** ClearTypeEnable;

*Enable the ClearType font display*
- under Windows 2000, standard font smoothing is forced, since Clear Type was introduced with XP

**function** CreateAnIcon (const Name, Description, Path, Parameters, WorkingDir, IconFilename: TFileName; const IconIndex: Integer; const RunMinimized: Boolean = false): TFileName;

*Create an Icon*
- return the .lnk file name (i.e. Name+.lnk')

**procedure** DrawCheckBox(hWnd: THandle; Handle: HDC; const Rect: TRect; Checked: boolean);

*Draw a CheckBox in the Canvas Handle of the Window hWnd, in the middle of the Rect coordinates*
- use theming under XP, Vista and Seven

**procedure** FillStringGrid(Source: TSQLTable; Dest: TStringGrid; Client: TSQLRest=nil);

*Fill TStringGrid.Cells[] with the supplied data*
- will be slower than the TSQLTableToGrid method, but will work on a non standard TDrawGrid component
- it will display date & time and enumerates as plain text, and handle the header properly (using the current mORMoti18n.pas language settings, if any)
- the Client optional parameter will be used to display any RecordRef column
- all data will be stored within the TStringGrid: you can safely release the Source data after having called this procedure

**function** GetShellFolderPath(const FolderID: Integer): string;

*Get the corresponding windows folder, from its ID*
procedure HideAppFormTaskBarButton;
  Low level VCL routine in order to hide the application from Windows task bar
  - don't use it directly: it's called by TVistaForm.CreateParams()

definition IsClearTypeEnabled: boolean;
  Test if the ClearType is enabled for font display
  - ClearType is a software technology that improves the readability of text on liquid crystal display (LCD) monitors

procedure Register;
  Register the TSynIntegerLabeledEdit component in the IDE toolbar
  - not necessary for the mORMot framework to run: since all User Interface is created from code, and not from the Delphi IDE, you don't have to register anything unless you define your own forms including those components
### 27.68. mORMotUIEdit.pas unit

**Purpose:** Record edition dialog, used to edit record content with mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

#### Units used in the mORMotUIEdit unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td>mORMoti18n</td>
<td>Internationalization (i18n) routines and classes for mORMot</td>
<td>2322</td>
</tr>
<tr>
<td>mORMotToolBar</td>
<td>ORM-driven Office 2007 Toolbar for mORMot</td>
<td>2390</td>
</tr>
<tr>
<td>mORMotUI</td>
<td>Grid to display database content for mORMot</td>
<td>2404</td>
</tr>
<tr>
<td>mORMotUILogin</td>
<td>Some common User Interface functions and dialogs for mORMot</td>
<td>2418</td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td>SynTaskDialog</td>
<td>Implement TaskDialog window (native on Vista/Seven, emulated on XP)</td>
<td>1824</td>
</tr>
</tbody>
</table>

#### mORMotUIEdit class hierarchy

```
TRTTIForm -> TRecordEditForm -> TVistaForm
```

#### Objects implemented in the mORMotUIEdit unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRecordEditForm</td>
<td>Record edition dialog, used to edit record content on the screen</td>
<td>2416</td>
</tr>
<tr>
<td>TRTTIForm</td>
<td>A common ancestor, used by both TRecordEditForm and TOptionsForm</td>
<td>2416</td>
</tr>
</tbody>
</table>
TRTTIForm = class(TVistaForm)
A common ancestor, used by both TRecordEditForm and TOptionsForm

OnCaptionName: TOncaptionName;
This event is used to customize screen text of property names

OnComponentCreate: TOncomponentCreate;
This event is used to customize the input components creation
- this event is also triggered once at the creation of the Option window, with Obj=Prop=nil and Parent=TOptionsForm: the event must call method Parent.AddEditors() / Parent.SetRecord() to add fields to the Option (this is not mandatory to the Record Edit window)
- this event is triggered once for every object, with Prop=nil, and should return nil if the object is to be added to the dialog, and something not nil if the object is to be ignored (same as a runtime-level _Name object)
- this is the only mandatory event of this component, for TOptionsForm
- this event is not mandatory for TRecordEditForm (you can call its SetRecord method directly)

OnComponentCreated: TOncomponentCreated;
This event is used to customize the input components after creation
- triggered when the component has been created
- can be used to disabled the component if user don’t have the right to modify its value; but he/she will still be able to view it

TRecordEditForm = class(TRTTIForm)
Record edition dialog, used to edit record content on the screen
- the window content is taken from the RTTI of the supplied record; all the User Interface (fields, etc...) is created from the class definition using RTTI: published properties are displayed as editing components
- caller must initialize some events, OnComponentCreate at least, in order to supply the objects to be added on the form
- components creation is fully customizable by some events
procedure SetRecord(aClient: TSQLRestClient; aRecord: TSQLRecord; CSVFieldNames: PUTF8Char=nil; Ribbon: TSQLRibbon=nil; FieldHints: string=''; FieldNamesWidth: integer=0; aCaption: string='');

Create the corresponding components on the dialog for editing a Record
- to be used by OnComponentCreate(nil,nil,EditForm) in order to populate the object tree of this Form
- create field on the window for all published properties of the supplied TSQLRecord instance
- properties which name starts by '_' are not added to the UI window
- user can customize the component creation by setting the OnComponentCreate / OnComponentCreated events
- the supplied aRecord instance must be available during all the dialog window modal apparition on screen
- by default, all published fields are displayed, but you can specify a CSV list in the optional CSVFieldNames parameter
- editor parameters are taken from the optional Ribbon parameter, and its EditFieldHints/EditExpandFieldHints/EditFieldNameWidth properties
- if Ribbon is nil, FieldHints may contain the hints to be displayed on screen (useful if your record is not stored in any TSQLRestClient, but only exists in memory); you can set FieldNamesWidth by hand in this case

property Client: TSQLRestClient read fClient;
The associated database Client, used to access remote data

property OnComponentValidate: TOnComponentValidate read fOnComponentValidate write fOnComponentValidate;
Event called to check if the content of a field on form is correct
- is checked when the user press the "Save" Button
- if returns false, component is focused and window is not closed

property Rec: TSQLRecord read fRec;
The associated Record to be edited

Types implemented in the mORMotUIEdit unit

TOnComponentCreate = function(Obj: TObject; Prop: TSQLPropInfo; Parent: TWinControl): TWinControl of object;
Event used for the window creation

TOnComponentCreated = procedure(Obj: TObject; Prop: TSQLPropInfo; Comp: TWinControl) of object;
Event used to customize the input component after creation

TOnComponentValidate = function(EditControl: TWinControl; Prop: TSQLPropInfo): boolean of object;
Event used for individual field validation
- must return TRUE if the specified field is correct, FALSE if the content is to be modified
- it's up to the handler to inform the user that this field is not correct, via a popup message for instance
- you should better use the TSQLRecord.AddFilterOrValidate() mechanism, which is separated from the UI (better multi-tier architecture)
27.69. mORMotUILogin.pas unit

**Purpose:** Some common User Interface functions and dialogs for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the mORMotUILogin unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1899</td>
</tr>
<tr>
<td>mORMotUI</td>
<td>Grid to display database content for mORMot&lt;br&gt;- this unit is a part of the freeware mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2404</td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
<tr>
<td>SynGdiPlus</td>
<td>GDI+ library API access&lt;br&gt;- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic&lt;br&gt;- make available most useful GDI+ drawing methods&lt;br&gt;- allows Antialiased rending of any EMF file using GDI+&lt;br&gt;- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1350</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features&lt;br&gt;- as a complement to SynCommons, which tended to increase too much&lt;br&gt;- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1721</td>
</tr>
<tr>
<td>SynTaskDialog</td>
<td>Implement TaskDialog window (native on Vista/Seven, emulated on XP)&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1824</td>
</tr>
</tbody>
</table>

**Objects implemented in the mORMotUILogin unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLoginForm</td>
<td>Form used to Log User and enter its password</td>
<td>2418</td>
</tr>
</tbody>
</table>

mORMotUILogin.pas unit - Rev. 1.18  Page 2419 of 2552
class function Login(const aTitle, aText: string; var aUserName, aPassWord: string; AllowUserNameChange: boolean; const CSVComboValues: string): boolean;
    Display the Login dialog window

class function PassWord(const aTitle, aText: string; var aPassWord: string): boolean;
    Display the password dialog window

class procedure OnIdleProcess(Sender: TSynBackgroundThreadAbstract; ElapsedMS: Integer);
    TOnIdleSQLRestClient-like event to process Windows Messages
    - to be assigned e.g. to TSQLRestClientURI.OnIdle property
    - global OnIdleProcessCursorChangeTimeout variable is used to display the crHourGlass cursor after a given time elapsed

class procedure OnIdleProcessForm(Sender: TSynBackgroundThreadAbstract; ElapsedMS: Integer);
    TOnIdleSQLRestClient-like event to process Windows Messages and write a temporary form on screen if it takes too long
    - to be assigned e.g. to TSQLRestClientURI.OnIdle property
    - global OnIdleProcessCursorChangeTimeout variable is used to display the crHourGlass cursor after a given time elapsed
    - global OnIdleProcessTemporaryFormTimeout variable is used to display a temporary form after a given time elapsed

Functions or procedures implemented in the mORMotUILogin unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choose</td>
<td>Ask the User to choose between some Commands</td>
<td>2420</td>
</tr>
<tr>
<td>CreateTempForm</td>
<td>Popup a temporary form with a message over all forms</td>
<td>2420</td>
</tr>
<tr>
<td>EnsureSingleInstance</td>
<td>Ensure that the program is launched once</td>
<td>2420</td>
</tr>
<tr>
<td>HtmlEscape</td>
<td>Convert an error message into html compatible equivalency</td>
<td>2420</td>
</tr>
<tr>
<td>InputBox</td>
<td>Ask the User to enter some string value</td>
<td>2420</td>
</tr>
<tr>
<td>InputQuery</td>
<td>Ask the User to enter some string value</td>
<td>2420</td>
</tr>
<tr>
<td>InputSelect</td>
<td>Ask the User to select one item from an array of strings</td>
<td>2420</td>
</tr>
<tr>
<td>InputSelectEnum</td>
<td>Ask the User to select one enumerate item</td>
<td>2421</td>
</tr>
<tr>
<td>SetStyle</td>
<td>Set the style for a form and a its buttons</td>
<td>2421</td>
</tr>
<tr>
<td>ShowException</td>
<td>Show an error dialog box, corresponding to a specified exception</td>
<td>2421</td>
</tr>
<tr>
<td>ShowLastClientError</td>
<td>Show an error dialog box, with the corresponding Client-Side information</td>
<td>2421</td>
</tr>
<tr>
<td>ShowMessage</td>
<td>Show an (error) message, using a Vista-Style dialog box</td>
<td>2421</td>
</tr>
</tbody>
</table>
### Functions or procedures

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShowMessage</td>
<td>Show an (error) message, using a Vista-Style dialog box</td>
<td>2421</td>
</tr>
<tr>
<td>YesNo</td>
<td>Ask the User to choose Yes or No [and Cancel], using a Vista-Style dialog box</td>
<td>2421</td>
</tr>
</tbody>
</table>

#### function `Choose(const aTitle, aCSVContent: string): integer; overload;`  
*Ask the User to choose between some Commands*  
- return the selected command index, starting numerotation at 100  
- this overloaded function expect the Content and the Commands to be supplied as CSV string (Content as first CSV, then commands)

#### function `Choose(const aTitle, aContent, aFooter: string; const Commands: array of string; aFooterIcon: TTaskDialogFooterIcon=tfiInformation): integer; overload;`  
*Ask the User to choose between some Commands*  
- return the selected command index, starting numerotation at 100

#### function `CreateTempForm(const aCaption: string; aPanelReference: PTPanel=nil; ScreenCursorHourGlass: boolean=false; aCaptionColor: integer=clNavy; aCaptionSize: integer=12): TForm;`  
*Popup a temporary form with a message over all forms*  
- caller should execute result.Free and reset the cursor if needed

#### procedure `EnsureSingleInstance;`  
*Ensure that the program is launched once*  
- the main project .dpr source file must contain:  
```pascal  
begin  
  Application.Initialize;  
  EnsureSingleInstance;  // program is launched once  
  Application.CreateForm(TMainForm, MainForm);  
  ....  
```

#### function `HtmlEscape(const Msg: string): string;`  
*Convert an error message into html compatible equivalency*  
- allow to display `<` & `&` correctly

#### function `InputBox(const ACaption, APrompt, ADefault: string; QueryMasked: boolean=false): string;`  
*Ask the User to enter some string value*  
- if QueryMasked=TRUE, will mask the prompt with '*' chars (e.g. for entering a password)

#### function `InputQuery(const ACaption, APrompt: string; var Value: string; QueryMasked: boolean=false): Boolean;`  
*Ask the User to enter some string value*  
- if QueryMasked=TRUE, will mask the prompt with '*' chars (e.g. for entering a password)

#### function `InputSelect(const ACaption, APrompt, AItemsText, ASelectedText: string): integer;`  
*Ask the User to select one item from an array of strings*  
- return the selected index, -1 if Cancel button was pressed
function InputSelectEnum(const ACaption, APrompt: string; EnumTypeInfo: PTypeInfo; var Index): boolean;
    Ask the User to select one enumerate item
    - use internally TEnumType.GetCaption() to retrieve the text to be displayed
    - Index must be an instance of this enumeration type (internally mapped to a PByte)

procedure SetStyle(Form: TComponent);
    Set the style for a form and its buttons
    - set the Default Font for all components, i.e. Calibri if available

function ShowException(E: Exception; CommonButtons: TCommonButtons=[cbOk]; const ContextMessage: string=''): integer;
    Show an error dialog box, corresponding to a specified exception

function ShowLastError(Client: TSQLRestClientURI; const ContextMessage: string=''; CommonButtons: TCommonButtons=[cbOk]): integer;
    Show an error dialog box, with the corresponding Client-Side information
    - retrieve last error message from Client.LastError* values

procedure ShowMessage(const Msg: string; Error: boolean=false); overload;
    Show an (error) message, using a Vista-Style dialog box

procedure ShowMessage(const Msg, Inst: string; Error: boolean=false); overload;
    Show an (error) message, using a Vista-Style dialog box

function YesNo(const aQuestion: string; const aConfirm: string=''; withCancel: boolean=true; Warning: boolean=false): integer;
    Ask the User to choose Yes or No [and Cancel], using a Vista-Style dialog box

Variables implemented in the mORMotUILogin unit

OnIdleProcessCursorChangeTimeout: integer = 100;
    Define when TLoginForm.OnIdleProcess() has to display the crHourGlass cursor after a given time elapsed, in milliseconds
    - default is 100 ms

OnIdleProcessTemporaryFormMessage: string;
    Define the message text displayed by TLoginForm.OnIdleProcessForm()
    - default is sOnIdleProcessFormMessage resourcestring, i.e. 'Please wait...'

OnIdleProcessTemporaryFormTimeout: integer = 2000;
    Define when TLoginForm.OnIdleProcessForm() has to display the temporary form after a given time elapsed, in milliseconds
    - default is 2000 ms, i.e. 2 seconds
27.70. mORMotUIOptions.pas unit

**Purpose:** General Options setting dialog for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the *mORMotUIOptions* unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>mORMot</em></td>
<td>Common ORM and SOA classes for mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1899</td>
</tr>
<tr>
<td><em>mORMoti18n</em></td>
<td>Internationalization (i18n) routines and classes for mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2322</td>
</tr>
<tr>
<td><em>mORMotToolBar</em></td>
<td>ORM-driven Office 2007 Toolbar for mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2390</td>
</tr>
<tr>
<td><em>mORMotUI</em></td>
<td>Grid to display database content for mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2404</td>
</tr>
<tr>
<td><em>mORMotUIEdit</em></td>
<td>Record edition dialog, used to edit record content with mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2415</td>
</tr>
<tr>
<td><em>mORMotUILogin</em></td>
<td>Some common User Interface functions and dialogs for mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2418</td>
</tr>
<tr>
<td><em>SynCommons</em></td>
<td>Common functions used by most Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
<tr>
<td><em>SynTaskDialog</em></td>
<td>Implement TaskDialog window (native on Vista/Seven, emulated on XP) - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1824</td>
</tr>
</tbody>
</table>

**Diagram:**

![Diagram of mORMotUIOptions class hierarchy]

**Objects implemented in the *mORMotUIOptions* unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOptionsForm</td>
<td>Options setting dialog</td>
<td>2423</td>
</tr>
</tbody>
</table>
TOptionsForm = class(TRTTIForm)

Options setting dialog  
- the settings parameters are taken from the RTTI of supplied objects: all the user interface is created from the code definition of classes; a visual tree node will reflect the properties recursion, and published properties are displayed as editing components  
- published textual properties may be defined as generic RawUTF8 or as generic string (with some possible encoding issue prior to Delphi 2009)  
- caller must initialize some events, OnComponentCreate at least, in order to supply the objects to be added on the form  
- components creation is fully customizable by some events  

SelectedNodeObjectOnShow: TObject;  
Creator may define this property to force a particular node to be selected at form showing

function AddEditors(Node: TTreeNode; Obj: TObject; const aCustomCaption: string=''; const aTitle: string=''): TTreeNode;  
Create corresponding nodes and components for updating Obj  
- to be used by OnComponentCreate(nil,nil,OptionsForm) in order to populate the object tree of this Form  
- properties which name starts by '_' are not added to the UI window  
- published properties of parents of Obj are also added

procedure AddToolbars(Scroll: TScrollBox; const aToolBarName: string; aEnum: PTypeInfo; const aActionHints: string; aActionsBits: pointer; aProp: PPropInfo; Obj: TObject);  
Create corresponding checkboxes lists for a given action toolbar  
- aEnum points to the Action RTTI  
- aActionHints is a multi line value containing the Hint captions for all available Actions  
- if aActionsBits is not nil, its bits indicates the Buttons to appear in the list
27.71. mORMotUIQuery.pas unit

**Purpose:** Form handling queries to a User Interface Grid for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the *mORMotUIQuery* unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>mORMot</em></td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td><em>mORMot</em>18n</td>
<td>Internationalization (i18n) routines and classes for mORMot</td>
<td>2322</td>
</tr>
<tr>
<td><em>mORMotUI</em></td>
<td>Grid to display database content for mORMot</td>
<td>2404</td>
</tr>
<tr>
<td><em>mORMotUILogin</em></td>
<td>Some common User Interface functions and dialogs for mORMot</td>
<td>2418</td>
</tr>
<tr>
<td><em>SynCommons</em></td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td><em>SynTaskDialog</em></td>
<td>Implement TaskDialog window (native on Vista/Seven, emulated on XP)</td>
<td>1824</td>
</tr>
</tbody>
</table>

**Objects implemented in the *mORMotUIQuery* unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TQueryForm</td>
<td>This Form perform simple Visual queries to a Grid</td>
<td>2425</td>
</tr>
</tbody>
</table>
TQueryForm = class(TVistaForm)

This Form perform simple Visual queries to a Grid
- mark or unmark items, depending of the input of the User on this form
- use TSQLRest.QueryIsTrue() method for standard fields and parameters
- use TSQLQueryCustom records previously registered to the TSQLRest class, by the TSQLRest.QueryAddCustom() method, to add some custom field search (e.g. to search into fields not available on the grid, or some data embedded inside a field - like .INI-like section entries)
- in practice, the query is very fast (immediate for standard fields and parameters), but can demand some bandwith for custom field search (since data has to be retrieved from the server to search within)

constructor Create(aOwner: TComponent; aTableToGrid: TSQLTableToGrid);
reintroduce;

Create the window instance
- all parameters (especially TSQLRest instance to use for custom search) are retrieved via the supplied TSQLTableToGrid
- caller must have used TSQLRest.QueryAddCustom() method to register some custom queries, if necessary
### 27.72. mORMotVCL.pas unit

**Purpose:** DB VCL dataset using TSQLTable/TSQLTableJSON data access
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

#### Units used in the mORMotVCL unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1899</td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
<tr>
<td>SynVirtualDataSet</td>
<td>DB VCL read-only virtual dataset&lt;br&gt;- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1840</td>
</tr>
</tbody>
</table>

#### mORMotVCL class hierarchy

- **TSynVirtualDataSet**: Read-only virtual TDataSet able to access a TSQLTable
  - **TSynSQLTableDataSet**: Read-only virtual TDataSet able to access a TSQLTable

#### Objects implemented in the mORMotVCL unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDBFieldDef</td>
<td>Store low-level DB.pas field information</td>
<td>2428</td>
</tr>
<tr>
<td>TSynSQLTableDataSet</td>
<td>Read-only virtual TDataSet able to access a TSQLTable</td>
<td>2426</td>
</tr>
</tbody>
</table>

**mORMotVCL.pas unit class hierarchy**

- **TSynVirtualDataSet** = **class(TSynVirtualDataSet)**

**Read-only virtual TDataSet able to access a TSQLTable**

**constructor** Create(Owner: TComponent; Table: TSQLTable; ForceWideString: boolean=false); **reintroduce**;

**Initialize the virtual TDataSet from a TSQLTable**
- WARNING: the supplied TSQLTable instance shall remain available all the time the returned TSynSQLTableDataSet instance is used, unless the TableShouldBeFreed property is set to true or CreateOwnedTable() constructor is used instead
- with non-Unicode version of Delphi, you can set ForceWideString to force the use of WideString fields instead of AnsiString, if needed
- the TDataSet will be opened once created
constructor CreateFromJSON(Owner: TComponent; const JSON: RawUTF8; const ColumnTypes: array of TSQLFieldType; ForceWideString: boolean=false); reintroduce; overload;

Initialize the virtual TDataSet from a supplied JSON result
- you can set the expected column types matching the results column layout
- this constructor will parse the supplied JSON content and create an internal TSQLTableJSON instance to process the data
- with non-Unicode version of Delphi, you can set ForceWideString to force the use of WideString fields instead of AnsiString, if needed
- the TDataSet will be opened once created

constructor CreateFromJSON(Owner: TComponent; const JSON: RawUTF8; const Tables: array of TSQLRecordClass; ForceWideString: boolean=false); reintroduce; overload;
Initialize the virtual TDataSet from a supplied JSON ORM result
- you can set the TSQLRecord classes to retrieve the expected column types
- this constructor will parse the supplied JSON content and create an internal TSQLTableJSON instance to process the data
- with non-Unicode version of Delphi, you can set ForceWideString to force the use of WideString fields instead of AnsiString, if needed
- the TDataSet will be opened once created

constructor CreateFromJSON(Owner: TComponent; const JSON: RawUTF8; ForceWideString: boolean=false); reintroduce; overload;
Initialize the virtual TDataSet from a supplied JSON result
- this constructor will parse the supplied JSON content and create an internal TSQLTableJSON instance to process the data, guessing the column types from the JSON content
- with non-Unicode version of Delphi, you can set ForceWideString to force the use of WideString fields instead of AnsiString, if needed
- the TDataSet will be opened once created

constructor CreateOwnedTable(Owner: TComponent; Table: TSQLTable; ForceWideString: boolean=false); reintroduce;
Initialize the virtual TDataSet owning a TSQLTable
- this constructor will set TableShouldBeFreed to TRUE
- with non-Unicode version of Delphi, you can set ForceWideString to force the use of WideString fields instead of AnsiString, if needed
- the TDataSet will be opened once created

destructor Destroy; override;
Finalize the class instance

property Table: TSQLTable read fTable write fTable;
Access to the internal TSQLTable[JSON] data
- you can use e.g. the SortFields() methods
- you may change the table content on the fly, if the column remains the same
property TableShouldBeFreed: boolean read fTableShouldBeFreed write fTableShouldBeFreed;

If the supplied TSQLTable instance should be released with this class
- Create() will left to FALSE (meaning that the TSQLTable instance shall remain available all the
  time the TSynSQLTableDataSet instance is used)
- CreateOwnedTable() will set to TRUE if you want the TSQLTable to be freed when this
  TSynSQLTableDataSet instance will be released
- you can also set it after Create(), on purpose

TDBFieldDef = record
  Store low-level DB.pas field information
  - as used by GetDBFieldDef and GetDBFieldValue

Functions or procedures implemented in the mORMotVCL unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetDBFieldDef</td>
<td>Get low-level DB.pas field information</td>
<td>2428</td>
</tr>
<tr>
<td>GetDBFieldValue</td>
<td>Fill a DB.pas field content</td>
<td>2428</td>
</tr>
<tr>
<td>JSONTableToDataSet</td>
<td>Convert a JSON ORM result into a VCL DataSet, following TSQLRecord field types</td>
<td>2428</td>
</tr>
<tr>
<td>JSONToDataSet</td>
<td>Convert a JSON result into a VCL DataSet, guessing the field types from the JSON</td>
<td>2429</td>
</tr>
<tr>
<td>JSONToDataSet</td>
<td>Convert a JSON result into a VCL DataSet, with a given set of column types</td>
<td>2429</td>
</tr>
</tbody>
</table>

procedure GetDBFieldDef(aTable: TSQLTable; aField: integer; out DBFieldDef: TDBFieldDef; aForceWideString: boolean=false);

Get low-level DB.pas field information
- ready to be added to a TDataSet as:
  aDataSet.FieldDefs.Add(FieldName,DBType,DBSize);

procedure GetDBFieldValue(aTable: TSQLTable; aRow: integer; aField: TField; aDataSet: TDataSet; const DBFieldDef: TDBFieldDef);

Fill a DB.pas field content
- used e.g. by mORMotMidasVCL.ToClientDataSet

function JSONTableToDataSet(aOwner: TComponent; const aJSON: RawUTF8; const Tables: array of TSQLRecordClass ; aForceWideString: boolean=false): TSynSQLTableDataSet;

Convert a JSON ORM result into a VCL DataSet, following TSQLRecord field types
- this function is just a wrapper around TSynSQLTableDataSet.CreateFromJSON()
- with non-Unicode version of Delphi, you can set aForceWideString to force the use of WideString fields instead of AnsiString, if needed
- with Unicode version of Delphi (2009+), string/UnicodeString will be used
function JSONToDataSet(aOwner: TComponent; const aJSON: RawUTF8; const ColumnTypes: array of TSQLFieldType; aForceWideString: boolean=false): TSynSQLTableDataSet;

Convert a JSON result into a VCL DataSet, with a given set of column types
- this function is just a wrapper around TSynSQLTableDataSet.CreateFromJSON()
- with non-Unicode version of Delphi, you can set aForceWideString to force the use of WideString fields instead of AnsiString, if needed
- with Unicode version of Delphi (2009+), string/UnicodeString will be used

function JSONToDataSet(aOwner: TComponent; const aJSON: RawUTF8; aForceWideString: boolean=false): TSynSQLTableDataSet;

Convert a JSON result into a VCL DataSet, guessing the field types from the JSON
- this function is just a wrapper around TSynSQLTableDataSet.CreateFromJSON()
- with non-Unicode version of Delphi, you can set aForceWideString to force the use of WideString fields instead of AnsiString, if needed
- with Unicode version of Delphi (2009+), string/UnicodeString will be used
27.73. mORMotWrappers.pas unit

**Purpose**: Generate cross-platform clients code and documentation from a mORMot server
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the mORMotWrappers unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1899</td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
<tr>
<td>SynLZ</td>
<td>SynLZ Compression routines - licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1393</td>
</tr>
<tr>
<td>SynMustache</td>
<td>Logic-less mustache template rendering - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1450</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features - as a complement to SynCommons, which tended to increase too much - licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1721</td>
</tr>
</tbody>
</table>

**Types implemented in the mORMotWrappers unit**

```
TOnCommandLineCall = procedure(aOptions: TServiceClientCommandLineOptions; const aService: TInterfaceFactory; aMethod: PServiceMethod; var aParams: TSQLRestURIParams) of object;

Event handler to let ExecuteFromCommandLine call a remote server
- before call, aParams.InBody will be set with the expected JSON content

TServiceClientCommandLineOptions = set of (cloPrompt, cloNoColor, cloPipe, cloHeaders, cloVerbose, cloNoExpand, cloNoBody);

The options retrieved during a ExecuteFromCommandLine() call
```

**Constants implemented in the mORMotWrappers unit**

```
EXECUTEFROMCOMMANDLINEHELP = ' % help -> show all services (interfaces)''#13#10 + ' % [service] [help] -> show all methods of a given service''#13#10 + ' % [service] [method] help -> show parameters of a given method ' + ' % [options] [service] [method] [parameters] -> call a given method ' + ' with [parameters] being name=value or name="value with spaces" or ' + ' name:{""some{":""json"}'} + ' and [options] as /nocolor /pipe /headers /verbose /noexpand /nobody';

Help information displayed by ExecuteFromCommandLine() with no command

WRAPPER_RESOURCENAME = 'WrappersDescription';

Internal Resource name used for bounded description
- as generated by FillDescriptionFromSource/ResourceDescriptionFromSource
- would be used e.g. by TWrapperContext.Create to inject the available text description from any
### Functions or procedures implemented in the `mORMotWrappers` unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddToServerWrapperMethod</td>
<td>You can call this procedure to add a 'Wrapper' method-based service to a given server, to allow code-generation of an ORM and SOA client</td>
<td>2432</td>
</tr>
<tr>
<td>ComputeFPCInterfacesUnit</td>
<td>You can call this procedure to generate the <code>mORMotInterfaces.pas</code> unit needed to register all needed interface RTTI for FPC</td>
<td>2432</td>
</tr>
<tr>
<td>ComputeFPCServerUnit</td>
<td>You can call this procedure to generate the <code>mORMotServer.pas</code> unit needed to compile a given server source code using FPC</td>
<td>2432</td>
</tr>
<tr>
<td>ContextFromMethod</td>
<td>Compute the information of an interface method, ready to be exported as JSON</td>
<td>2432</td>
</tr>
<tr>
<td>ContextFromMethods</td>
<td>Compute the information of an interface, ready to be exported as JSON</td>
<td>2432</td>
</tr>
<tr>
<td>ContextFromModel</td>
<td>Compute the Model information, ready to be exported as JSON</td>
<td>2433</td>
</tr>
<tr>
<td>ExecuteFromCommandLine</td>
<td>Command-line SOA remote access to mORMot interface-based services</td>
<td>2433</td>
</tr>
<tr>
<td>FillDescriptionFromSource</td>
<td>Rough parsing of the supplied <code>.pas</code> unit, adding the /// commentaries into a TDocVariant content</td>
<td>2433</td>
</tr>
<tr>
<td>GenerateAsynchServices</td>
<td>This function would generate a pascal unit defining asynchronous (non-blocking) types from a DDD's blocking dual-phase Select/Command service</td>
<td>2433</td>
</tr>
<tr>
<td>ResourceDescriptionFromSource</td>
<td>Rough parsing of the supplied <code>.pas</code> unit, adding the /// commentaries into a compressed binary resource</td>
<td>2434</td>
</tr>
<tr>
<td>WrapperFakeServer</td>
<td>Instantiate a TSQLRest server instance, including supplied ORM and SOA definitions</td>
<td>2434</td>
</tr>
<tr>
<td>WrapperForPublicAPI</td>
<td>Generate a code/doc wrapper for a given set of types and Mustache template content</td>
<td>2434</td>
</tr>
<tr>
<td>WrapperFromModel</td>
<td>Generate a code/doc wrapper for a given Model and Mustache template content</td>
<td>2435</td>
</tr>
<tr>
<td>WrapperMethod</td>
<td>You can call this procedure within a method-based service allow code-generation of an ORM and SOA client from a web browser</td>
<td>2435</td>
</tr>
</tbody>
</table>
procedure AddToServerWrapperMethod(Server: TSQLRestServer; const Path: array of TFileName; const SourcePath: TFileName='');

You can call this procedure to add a 'Wrapper' method-based service to a given server, to allow code-generation of an ORM and SOA client
- you have to specify one or several client *.mustache file paths
- the first path containing any *.mustache file will be used as templates
- if no path is specified (i.e. as []), it will search in the .exe folder
- the root/wrapper URI will be accessible without authentication (i.e. from any plain browser)
- for instance:
  aServer := TSQLRestServerFullMemory.Create(aModel, 'test.json', false, true);
  AddToServerWrapperMethod(aServer, ['..']);
- optional SourcePath parameter may be used to retrieve additional description from the comments of the source code of the unit

procedure ComputeFPCTemplateUnit(const Path: array of TFileName; DestFileName: TFileName='');

You can call this procedure to generate the mORMotTemplate.pas unit needed to register all needed interface RTTI for FPC
- to circumvent http://bugs.freepascal.org/view.php?id=26774 unresolved issue
- will locate FPC-mORMotInterfaces.pas.mustache in the given Path[] array
- will write the unit using specified file name or to mORMotInterfaces.pas in the current directory if DestFileName is '', or to a sub-folder of the matching Path[] if DestFileName starts with '\\' (to allow relative folder use)
- all used interfaces will be exported, including SOA and mocking/stubbing types: so you may have to run this function AFTER all process is done

procedure ComputeFPCServerUnit(Server: TSQLRestServer; const Path: array of TFileName; DestFileName: TFileName='');

You can call this procedure to generate the mORMotServer.pas unit needed to compile a given server source code using FPC
- will locate FPCServer-mORMotServer.pas.mustache in the given Path[] array
- will write the unit using specified file name or to mORMotServer.pas in the current directory if DestFileName is '', or to a sub-folder of the matching Path[] if DestFileName starts with '\\' (to allow relative folder use)
- the missing RTTI for records and interfaces would be defined, together with some patch comments for published record support (if any) for the ORM

function ContextFromMethod(const method: TServiceMethod): variant;

Compute the information of an interface method, ready to be exported as JSON
- to be used e.g. for the implementation of the MVC controller via interfaces
- no description text will be included - use ContextFromModel() if needed

function ContextFromMethods(int: TInterfaceFactory): variant;

Compute the information of an interface, ready to be exported as JSON
- to be used e.g. for the implementation of the MVC controller via interfaces
- no description text will be included - use ContextFromModel() if needed
function ContextFromModel(aServer: TSQLRestServer; const aSourcePath: TFileName=''; const aDescriptions: TFileName=''): variant;

Compute the Model information, ready to be exported as JSON
- will publish the ORM and SOA properties
- to be used e.g. for client code generation via Mustache templates
- optional aSourcePath parameter may be used to retrieve additional description from the comments of the source code of the unit - this text content may also be injected by WRAPPER_RESOURCENAME
- you may specify a description file (as generated by FillDescriptionFromSource)

procedure ExecuteFromCommandLine(const aServices: array of TGUID; const aOnCall: TOnCommandLineCall; const aDescriptions: TFileName = '');

Command-line SOA remote access to mORMot interface-based services
- supports the following EXECUTEFROMCOMMANDLINEHELP commands
- you shall have registered the aServices interface(s) by a previous call to the overloaded Get(TypeInfo(IMyInterface)) method or RegisterInterfaces()
- you may specify an optional description file, as previously generated by mORMotWrappers' FillDescriptionFromSource function - a local 'WrappersDescription' resource will also be checked
- to actually call the remote server, aOnGetClient should be supplied

procedure FillDescriptionFromSource(var Descriptions: TDocVariantData; const SourceFileName: TFileName);

Rough parsing of the supplied .pas unit, adding the /// commentaries into a TDocVariant content

function GenerateAsynchServices(const services: array of TGUID; const queries: array of TClass; const units: array of const; const additionalcontext: array of const; Template, FileName, ProjectName, CallType, CallFunction, Key, KeyType, ExceptionType: RawUTF8; DefaultDelay: integer; const CustomDelays: array of const): RawUTF8;

This function would generate a pascal unit defining asynchronous (non-blocking) types from a DDD's blocking dual-phase Select/Command service
- you should specify the services to be converted, as an array - note that due to how RTTI is stored by the compiler, all "pure input" parameters should be defined explicitly as "const", otherwise the generated class won't match
- optionally, the TCQRSServiceClass implementing the first Select() phase of the blocking service may be specified in queries array; a set of unit names in which those TCQRSServiceClass are defined may be specified
- a Mustache template content should be provided - e.g. asynch.pas.mustache as published in SQLite3\DDD\dom folder of the source code repository
- FileName would contain the resulting unit filename (without the .pas)
- ProjectName would be written in the main unit comment
- CallType should be the type used at Domain level to identify each asynchronous call - this type should be an integer, or a function may be supplied as CallFunction (matching VariantToInteger signature)
- the first phase of the service should have set Key: KeyType, which would be used to create a single shared asynchronous service instance for all keys
- ExceptionType may be customize, mainly to use a Domain-specific class
- blocking execution may reach some timeout waiting for the asynchronous acknowledgement: a default delay (in ms) is to be supplied, and some custom delays may be specified as trios, e.g. ['IMyInterface', 'Method', 10000, ...]
function ResourceDescriptionFromSource(const ResourceDestFileName: TFileName; const SourceFileNames: array of TFileName; const JsonDestFileName: TFileName = ''): variant;

Rough parsing of the supplied .pas unit, adding the /// commentaries into a compressed binary resource
- could be then compiled into a WRAPPER_RESOURCE_NAME resource, e.g. via the following .rc source file, assuming ResourceDestFileName='wrapper.desc':
  WrappersDescription 10 "wrapper.desc"
- you may specify a .json file name, for debugging/validation purposes
- calls internally FillDescriptionFromSource
- returns the TDocVariant JSON object corresponding to all descriptions

function WrapperFakeServer(const aTables: array of TSQLRecordClass; const aRoot: RawUTF8; const aSharedServices: array of TGUID; const aSharedServicesContract: array of RawUTF8; aResultAsJSONObjectWithoutResult: boolean): TSQLRestServerFullMemory;

Instantiate a TSQLRest server instance, including supplied ORM and SOA definitions
- will use aTables[] to define the ORM information, and supplied aSharedServices[] aSharedServicesContract[] for SOA definition of a shared API, implemented as abstract classes using TInterfaceStub
- as used e.g. by WrapperForPublicAPI() to generate some code/doc wrappers

function WrapperForPublicAPI(const aTables: array of TSQLRecordClass; const aRoot, aMustacheTemplate, aFileName: RawUTF8; const aSharedServices: array of TGUID; const aSharedServicesContract: array of RawUTF8; aResultAsJSONObjectWithoutResult: boolean; aPort: Integer; aHelpers: TSynMustacheHelpers=nil; aContext: PVariant=nil; const aDescriptions: TFileName=''): RawUTF8;

Generate a code/doc wrapper for a given set of types and Mustache template content
- will use aTables[] to define the ORM information, and supplied aSharedServices[] aSharedServicesContract[] for SOA definition of a shared API (expected to be called from TSQLRestClientURI.ServiceDefineSharedAPI)
- aFileName will be transmitted as {{filename}}, e.g. 'mORMotClient'
- you should also specify a "fake" HTTP port e.g. 888
- the template content could be retrieved from a file via StringFromFile()
- you may optionally retrieve a copy of the data context as TDocVariant
- this function may be used to generate the client at build time, directly from a just built server, in an automated manner
- you may specify custom helpers (e.g. via TSynMustache.HelpersGetStandardList) and retrieve the generated data context after generation (if aContext is a TDocVariant object, its fields would be added to the rendering context), or a custom description file (as generated by FillDescriptionFromSource)
function WrapperFromModel(aServer: TSQLRestServer; const aMustacheTemplate, aFileName: RawUTF8; aPort: integer; aHelpers: TSynMustacheHelpers=nil; aContext: PVariant=nil; const aDescriptions: TFileName=''): RawUTF8;

Generate a code/doc wrapper for a given Model and Mustache template content
- will use all ORM and SOA properties of the supplied server
- aFileName will be transmitted as {{filename}}, e.g. 'mORMotClient'
- you should also specify a "fake" HTTP port e.g. 888
- the template content could be retrieved from a file via StringFromFile()
- you may optionally retrieve a copy of the data context as TDocVariant
- this function may be used to generate the client at build time, directly from a just built server, in
  an automated manner
- you may specify custom helpers (e.g. via TSynMustache.HelpersGetStandardList) and retrieve the
  generated data context after generation (if aContext is a TDocVariant object, its fields would be
  added to the rendering context), or a custom description file (as generated by
  FillDescriptionFromSource)

procedure WrapperMethod(Ctxt: TSQLRestServerURIContext; const Path: array of TFileName; const SourcePath: TFileName=''; const Descriptions: TFileName='');

You can call this procedure within a method-based service allow code-generation of an ORM and
SOA client from a web browser
- you have to specify one or several client *.mustache file paths
- the first path containing any *.mustache file will be used as templates
- for instance:

procedure TCustomServer.Wrapper(Ctxt: TSQLRestServerURIContext);
begin // search in the current path
  WrapperMethod(Ctxt,['.']);
end;

- optional SourcePath parameter may be used to retrieve additional description from the
  comments of the source code of the unit
- you may specify a description file (as generated by FillDescriptionFromSource)

Variables implemented in the mORMotWrappers unit

DESCRIPTION_ITEM_PREFIX: RawUTF8 = ' [*]';

How FillDescriptionFromSource() handles trailing '-' in parsed comments
- default is ['*'], as expected by buggy AsciiDoc format
## 27.74. TestSQL3FPCInterfaces.pas unit

**Purpose**: SOA interface methods definition to circumvent FPC missing RTTI
- generated at 2016-06-14 13:49:41

### Units used in the TestSQL3FPCInterfaces unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| dddDomAuthInterfaces       | Shared DDD Domains: Authentication objects and interfaces  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2437 |
| dddDomUserCQRS             | Shared DDD Domains: User CQRS Repository interfaces  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2442 |
| dddDomUserInterfaces       | Shared DDD Domains: User interfaces definition  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2445 |
| dddDomUserTypes            | Shared DDD Domains: User objects definition  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2447 |
| dddInfraEmailer            | Shared DDD Infrastructure: generic emailing service  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2471 |
| mORMot                     | Common ORM and SOA classes for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1899 |
| mORMotDDD                  | Domain-Driven-Design toolbox for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2285 |
| SynCommons                 | Common functions used by most Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 717  |
| SynLog                     | Logging functions used by Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1363 |
| SynSelfTests               | Automated tests for common units of the Synopse mORMot Framework  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1528 |
| SynTable                   | Filter/database/cache/buffer/security/search/multithread/OS features  
- as a complement to SynCommons, which tended to increase too much  
- licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1721 |
27.75. dddDomAuthInterfaces.pas unit

*Purpose:* Shared DDD Domains: Authentication objects and interfaces
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

### Units used in the dddDomAuthInterfaces unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>mORMot</em></td>
<td>Common ORM and SOA classes for mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1899</td>
</tr>
<tr>
<td><em>mORMotDDD</em></td>
<td>Domain-Driven-Design toolbox for mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2285</td>
</tr>
<tr>
<td><em>SynCommons</em></td>
<td>Common functions used by most Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
</tbody>
</table>

### Objects implemented in the dddDomAuthInterfaces unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>IDomAuthCommand</em></td>
<td>Repository service to update or register new authentication credentials</td>
<td>2438</td>
</tr>
<tr>
<td><em>IDomAuthQuery</em></td>
<td>Repository service to authenticate credentials via a dual pass challenge</td>
<td>2437</td>
</tr>
<tr>
<td><em>TAuthInfo</em></td>
<td>DDD entity used to store authentication information</td>
<td>2437</td>
</tr>
</tbody>
</table>

**TAuthInfo = class(TSynPersistent)**

*DDD entity used to store authentication information*

*property* LogonName: TAuthInfoName *read fLogonName write fLogonName;*

*The textual identifier by which the user would recognize himself*

**getIDAuthQuery = interface(ICQRSService)**

*Repository service to authenticate credentials via a dual pass challenge*

*function ChallengeSelectFinal(const aChallengedPassword: TAuthQueryNonce): TCQRSResult;*

*Validate the first phase of a dual pass challenge authentication*
function ChallengeSelectFirst(const aLogonName: RawUTF8): TAuthQueryNonce;
Initiate the first phase of a dual pass challenge authentication

function Get(out aAggregate: TAuthInfo): TCQRSResult;
Retrieve some information about the current selected credential

function Logged: boolean;
Returns TRUE if the dual pass challenge did succeed

function LogonName: RawUTF8;
Returns the logon name of the authenticated user

function SelectByName(const aLogonName: RawUTF8): TCQRSResult;
- this method execution will be disabled for most clients

IDomAuthCommand = interface(IDomAuthQuery)
Repository service to update or register new authentication credentials

function Add(const aLogonName: RawUTF8; aHashedPassword: TAuthQueryNonce): TCQRSResult;
Register a new credential, from its LogonName/HashedPassword values
- aHashedPassword should match the algorithm expected by the actual implementation class,
  over UTF-8 encoded LogonName+':'+Password
- on success, the newly created credential will be the currently selected

function Commit: TCQRSResult;
Write all pending changes prepared by Add/UpdatePassword/Delete methods

function Delete: TCQRSResult;
Delete the current selected credential
- this method execution will be disabled for most clients

function UpdatePassword(const aHashedPassword: TAuthQueryNonce): TCQRSResult;
Update the current selected credential password
- aHashedPassword should match the algorithm expected by the actual implementation class,
  over UTF-8 encoded LogonName+':'+Password
- will be allowed only for the current challenged user

Types implemented in the dddDomAuthInterfaces unit

TAuthQueryNonce = RawUTF8;
The data type which will be returned during a password challenge
- in practice, will be e.g. Base-64 encoded SHA-256 binary hash
27.76. **dddDomCountry.pas unit**

*Purpose:* Shared DDD Domains: TCountry object definition
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the dddDomCountry unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| mORMot    | Common ORM and SOA classes for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1899 |
| mORMotDDD | Domain-Driven-Design toolbox for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2285 |
| SynCommons | Common functions used by most Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 717 |
| SynTests  | Unit test functions used by Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1832 |

**Objects implemented in the dddDomCountry unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCountry</td>
<td>Defines a Country identifier object</td>
<td>2439</td>
</tr>
</tbody>
</table>

**TCountry = class(TSynPersistent)**

*Defines a Country identifier object*
- will store internally the country as 16-bit ISO 3166-1 numeric value
- see also some low-level class methods for direct values conversions with no persistence

*function* **Equals(another: TCountry): boolean; reintroduce;**

*Returns TRUE if both Country instances have the same content*
- slightly faster than global function ObjectEquals(self,another)

*class function* **FromAlpha2(const alpha: TCountryIsoAlpha2): TCountryIdentifier;**

*Low-level Country conversion from its alpha-2 code*
- returns ccUndefined if the supplied text has no case-insensitive match
class function FromAlpha3(const alpha: TCountryIsoAlpha3): TCountryIdentifier;
  Low-level Country conversion from its alpha-3 code
  - returns ccUndefined if the supplied Text has no case-insensitive match

class function FromEnglish(const text: RawUTF8): TCountryIdentifier;
  Low-level case-insensitive Country conversion from its plain English text
  - returns ccUndefined if the supplied Text has no case-insensitive match

class function FromIso(iso: TCountryIsoNumeric): TCountryIdentifier;
  Low-level Country conversion from its alpha-2 code
  - returns ccUndefined if the supplied 16-bit number as no match

class function ToAlpha2(id: TCountryIdentifier): TCountryIsoAlpha2;
  Low-level Country conversion into its alpha-2 code

class function ToAlpha3(id: TCountryIdentifier): TCountryIsoAlpha3;
  Low-level Country conversion into its alpha-3 code

class function ToEnglish(id: TCountryIdentifier): RawUTF8;
  Low-level Country conversion into its plain English text

class function ToIso(id: TCountryIdentifier): TCountryIsoNumeric;
  Low-level Country conversion to its ISO 3166-1 numeric 3-digit code

class procedure RegressionTests(test: TSynTestCase);
  Built-in simple unit tests

property Alpha2: TCountryIsoAlpha2 read GetIsoAlpha2 write SetIsoAlpha2;
  The ISO 3166-1 alpha-2 code of this country

property Alpha3: TCountryIsoAlpha3 read GetIsoAlpha3 write SetIsoAlpha3;
  The ISO 3166-1 alpha-3 code of this country

property English: RawUTF8 read GetEnglish;
  Plain English text of this country, e.g. 'France' or 'United States'

property Identifier: TCountryIdentifier read GetIdentifier write SetIdentifier;
  Internal enumerate corresponding to this country

property Iso: TCountryIsoNumeric read fIso write fIso;
  The stored and transmitted value is this ISO 3166-1 numeric 3-digit code

Types implemented in the dddDomCountry unit

TCountryIdentifier = ( ccUndefined, ccAF, ccAX, ccAL, ccDZ, ccAS, ccAD, ccAO, ccAI, ccAQ, ccAG, ccAR, ccAM, ccAW, ccAU, ccAT, ccAZ, ccBS, ccBH, ccBD, ccBY, ccBE, ccBZ, ccBJ, ccBM, ccBT, ccBO, ccBQ, ccBA, ccBW, ccBV, ccBR, ccIO, ccBN, ccBG, ccBF, ccBI, ccKH, ccCM, ccCA, ccCV, ccKY, ccCF, ccTD, ccCL, ccCN, ccX, ccCC, ccCO, ccKM, ccCG, ccCD, ccK, ccCR, ccCI, ccHR, ccCU, ccCW, ccCY, ccCZ, ccDK, ccDJ, ccDM, ccDO, ccEC, ccEG, ccSV, ccGQ, ccER, ccEE, ccET, ccFK, ccFO, ccFJ, ccFI, ccFR, ccG, ccPF, ccTF, ccGA, ccGM, ccGE, ccDE, ccGH, ccGI, ccGR, ccGL, ccGD, ccGP, ccGU, ccGT, ccGG, ccGN, ccGW, ccGY, ccHT, ccHM, ccVA, ccHN, cckH, cckU, cckS, ccIN, ccID, ccIR, ccIQ, ccIE, ccIM, ccIL, ccIT, ccJM, ccJP, ccJE, ccJO, cckZ, cckE, cckI, cckP, cckR, ccKW, ccKG, ccla, ccLV, ccLB, ccls, cclR, ccLY, ccli, cclT, cclU, cclO, ccmK, ccmG, ccmW, ccmY, ccmV, ccmL, ccMT, ccmH, ccmQ, ccmR, ccYT, ccMY, ccFM, ccMD, ccMC, ccmN, ccmE, ccmS, ccmA, ccmZ, cmmM, ccmN, ccmR, ccmP, ccmL, ccmC, ccmZ, ccmN, ccmI, ccmNE, ccmNG, ccmU, ccmNF, ccmMP, ccmNO, ccmOM, ccmPK, ccmPW, ccmPS, ccmPA, ccmPG, ccmPY, ccmP,
Country identifiers, following ISO 3166-1 standard

TCountryIsoAlpha2 = type RawUTF8;

Store ISO 3166-1 alpha-2 code

TCountryIsoAlpha3 = type RawUTF8;

Store ISO 3166-1 alpha-3 code

TCountryIsoNumeric = type word;

Store a ISO 3166-1 numeric value as 16-bit unsigned integer
27.77. dddDomUserCQRS.pas unit

**Purpose**: Shared DDD Domains: User CQRS Repository interfaces
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the dddDomUserCQRS unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dddDomUserTypes</td>
<td>Shared DDD Domains: User objects definition</td>
<td>2447</td>
</tr>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td>mORMotDDD</td>
<td>Domain-Driven-Design toolbox for mORMot</td>
<td>2285</td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
</tbody>
</table>

**Objects implemented in the dddDomUserCQRS unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDomUserCommand</td>
<td>Defines an abstract CQRS Repository for Writing TUser Aggregate Roots</td>
<td>2443</td>
</tr>
<tr>
<td>IDomUserQuery</td>
<td>Defines an abstract CQRS Repository for Reading TUser Aggregate Roots</td>
<td>2442</td>
</tr>
</tbody>
</table>

**IDomUserQuery = interface(ICQRSService)**

*Defines an abstract CQRS Repository for Reading TUser Aggregate Roots*
- this interface allows only read access to the Aggregate: see IDomUserCommand to modify the content
- you could use SelectByLogonName, SelectByLastName or SelectByEmailValidation methods to initialize a request, then call Get, GetAll or GetNext to retrieve the actual matching Aggregate Roots

```pascal
function Get(out aAggregate: TUser): TCQRSResult;

Retrieve a single TUser
```
function GetAll(out aAggregates: TUserObjArray): TCQRSResult;

Retrieve all matching TUser instances
- the caller should release all returned TUser by calling
  ObjArrayClear(aAggregates);

function GetCount: integer;

Retrieve how many TUser instances do match the selection

function GetNext(out aAggregate: TUser): TCQRSResult;

Retrieve the next matching TUser instances
- returns cqrsNoMoreData if there is no more pending data

function HowManyValidatedEmail: integer;

Retrieve how many TUser have their email validated

function SelectAll: TCQRSResult;

Would select all TUser instances
- you should not use this search criteria, since it may return a huge number of values
- then use GetCount, GetAll() or GetNext() methods to retrieve the items

function SelectByEmailValidation(aValidationState: TDomUserEmailValidation): TCQRSResult;

Would select one or several TUser from their email validation state
- then use GetCount, GetAll() or GetNext() methods to retrieve the items

function SelectByLastName(const aName: TLastName; aStartWith: boolean): TCQRSResult;

Would select one or several TUser from their last name
- will search for a full matching name, unless aStartWith is TRUE so that it would search for the beginning characters
- then use GetCount, GetAll() or GetNext() methods to retrieve the items

function SelectByLogonName(const aLogonName: RawUTF8): TCQRSResult;

Would select a single TUser from its logon name
- then use Get() method to retrieve its content

IDomUserCommand = interface(IDomUserQuery)

Defines an abstract CQRS Repository for Writing TUser Aggregate Roots
- would implement a dual-phase commit to change TUser content
- first phase consists in calling Add, Update, Delete or DeleteAll methods which would call the registered validators on the supplied content
- you can call Add, Update, Delete or DeleteAll methods several times, so that several write operations will be recorded for the TUser
- during the first phase, nothing is actually written to the persistence storage itself (which may be a RDBMS or a NoSQL engine)
- then the second phase would take place when the Commit method would be executed, which would save all prepared content to the actual storage engine (e.g. using a transaction via a BATCH process if implemented by mORMot’s ORM, via TInfraRepoUser as defined in dddInfraRepoUser)
function Add(const aAggregate: TUser): TCQRSResult;
   Persist a new TUser aggregate

function Commit: TCQRSResult;
   Write all pending changes prepared by Add/Update/Delete methods
   - following the dual-phase pattern, nothing would be written to the actual persistence store
     unless this method is actually called

function Delete: TCQRSResult;
   Erase an existing TUser aggregate
   - the existing content should have been retrieved by a previous Select* method, e.g.
     IDomUserQuery.SelectByLogonName

function DeleteAll: TCQRSResult;
   Erase existing TUser aggregate, matching a
   - the existing content should have been retrieved by a previous Select* method, e.g.
     IDomUserQuery.SelectByLogonName: a plain DeleteAll call with no prior Select* would return
     an error

function Rollback: TCQRSResult;
   Flush any pending changes prepared by Add/Update/Delete methods
   - is the same as releasing the actual IDomUserCommand instance and creating a new one

function Update(const aUpdatedAggregate: TUser): TCQRSResult;
   Update an existing TUser aggregate
   - the existing content should have been retrieved by a previous Select* method, e.g.
     IDomUserQuery.SelectByLogonName
27.78. dddDomUserInterfaces.pas unit

*Purpose*: Shared DDD Domains: User interfaces definition

- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the dddDomUserInterfaces unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dddDomUserTypes</td>
<td>Shared DDD Domains: User objects definition</td>
<td>2447</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMotDDD</td>
<td>Domain-Driven-Design toolbox for mORMot</td>
<td>2285</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

**Objects implemented in the dddDomUserInterfaces unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDomUserEmailCheck</td>
<td>Defines a service able to check the correctness of email addresses</td>
<td>2445</td>
</tr>
<tr>
<td>IDomUserEmailer</td>
<td>Defines a generic service able to send emails</td>
<td>2446</td>
</tr>
<tr>
<td>IDomUserEmailValidation</td>
<td>Defines a service sending a confirmation email to validate an email address</td>
<td>2446</td>
</tr>
<tr>
<td>IDomUserTemplate</td>
<td>Defines a service for generic rendering of a template</td>
<td>2446</td>
</tr>
</tbody>
</table>

**I DomUserEmailCheck = interface(IInvokable)**

*Defines a service able to check the correctness of email addresses*

- will be implemented e.g. by TDDDEmailServiceAbstract and TDDDEmailValidationService as defined in the dddInfraEmail unit
function CheckRecipient(const aEmail: RawUTF8): TCQRSResult;
  Check if the supplied email address seems correct

function CheckRecipients(const aEmails: TRawUTF8DynArray): TCQRSResult;
  Check if the supplied email addresses seem correct

IDomUserEmailValidation = interface(IDomUserEmailCheck)
  Defines a service sending a confirmation email to validate an email address
  - will be implemented e.g. by TDDDEmailValidationService as defined in the dddInfraEmail unit

function ComputeURIForReply(const aLogonName, aEmail: RawUTF8): RawUTF8;
  Internal method used to compute the validation URI
  - will be included as data context to the email template, to create the validation link

function StartEmailValidation(const aTemplate: TDomUserEmailTemplate; const aLogonName, aEmail: RawUTF8): TCQRSResult;
  Initiate an email validation process, using the given template

IDomUserEmailer = interface(IInvokable)
  Defines a generic service able to send emails
  - will be implemented e.g. by TDDDEmailerDaemon as defined in the dddInfraEmailer unit

IDomUserTemplate = interface(IInvokable)
  Defines a service for generic rendering of a template
  - will be implemented e.g. via our SynMustache engine by TDDDEemplateAbstract and TDDDEemplateFromFolder as defined in the dddInfraEmailer unit
27.79. **dddDomUserTypes.pas unit**

_Purpose_: Shared DDD Domains: User objects definition
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the **dddDomUserTypes** unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dddDomCountry</strong></td>
<td>Shared DDD Domains: TCountry object definition&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2439</td>
</tr>
<tr>
<td><strong>mORMot</strong></td>
<td>Common ORM and SOA classes for mORMot&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1899</td>
</tr>
<tr>
<td><strong>mORMotDDD</strong></td>
<td>Domain-Driven-Design toolbox for mORMot&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2285</td>
</tr>
<tr>
<td><strong>SynCommons</strong></td>
<td>Common functions used by most Synopse projects&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
<tr>
<td><strong>SynTests</strong></td>
<td>Unit test functions used by Synopse projects&lt;br&gt;- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1832</td>
</tr>
</tbody>
</table>

**Objects implemented in the **dddDomUserTypes** unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TAddress</strong></td>
<td>Address object</td>
<td>2448</td>
</tr>
<tr>
<td><strong>TDomUserEmailTemplate</strong></td>
<td>How a confirmation email is to be rendered, for email address validation</td>
<td>2448</td>
</tr>
</tbody>
</table>

**dddDomUserTypes class hierarchy**

```plaintext
TSynAutoCreateFields
   └── TAdresse
       └── TPerson
           └── TPersonContactable
               └── TUser

TSynPersistent
   ├── TPerson
   │    └── TPersonFullName
   │         └── TPersonBirthDate
   │               └── TDomUserEmailTemplate
   └── TAdresse
```

*dddDomUserTypes.pas unit - Rev. 1.18*
<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPerson</td>
<td>Person object</td>
<td>2448</td>
</tr>
<tr>
<td>TPersonBirthDate</td>
<td>Person birth date</td>
<td>2448</td>
</tr>
<tr>
<td>TPersonContactable</td>
<td>A Person object, with some contact information</td>
<td>2448</td>
</tr>
<tr>
<td>TPersonFullName</td>
<td>Person full name</td>
<td>2448</td>
</tr>
<tr>
<td>TUser</td>
<td>An application level-user, whose account would be authenticated per Email</td>
<td>2449</td>
</tr>
</tbody>
</table>

TAddress = class(TSynAutoCreateFields)

*Address object*
- we tried to follow a simple but worldwide layout - see
  http://en.wikipedia.org/wiki/Address_%28geography%29#Address_format

TPersonFullName = class(TSynPersistent)

*Person full name*

TPersonBirthDate = class(TSynPersistent)

*Person birth date*

TPerson = class(TSynAutoCreateFields)

*Person object*

TPersonContactable = class(TPerson)

*A Person object, with some contact information*
- an User is a person, in the context of an application

class procedure RegressionTests(test: TSynTestCase);
  *Built-in simple unit tests*

TDomUserEmailTemplate = class(TSynPersistent)

*How a confirmation email is to be rendered, for email address validation*
- this information will be available as data context, e.g. to the Mustache template used for rendering of the email body

property Application: RawUTF8 read fApplication write fApplication;
  *The name of the application, currently sending the confirmation*

property FileName: RawUTF8 read fFileName write fFileName;
  *The local file name of the Mustache template*

property Info: variant read fInfo write fInfo;
  *Any unstructured additional information, also supplied as data context*

property SenderEmail: RawUTF8 read fSenderEmail write fSenderEmail;
  *The "sender" field of the validation email*
property Subject: RawUTF8 read fSubject write fSubject;
    The "subject" field of the validation email

TUser = class(TPersonContactable)
    An application level-user, whose account would be authenticated per Email

property EmailValidated: TDomUserEmailValidation read fEmailValidated write fEmailValidated;
    Will reflect the current state of email validation process for this user
    - the validation is not handled by this class: this is just a property which reflects the state of
      TDDDEmailValidationService/IDomUserEmailValidation

property LogonName: TLogonName read fLogonName write fLogonName;
    The logon name would be the main entry point to the application

Types implemented in the dddDomUserTypes unit

TDomUserEmailValidation = ( evUnknown, evValidated, evFailed );
    The status of an email validation process
27.80. dddInfraApps.pas unit

**Purpose:** Shared DDD Infrastructure: Application/Daemon implementation classes
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

### Units used in the dddInfraApps unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dddInfraSettings</td>
<td>Shared DDD Infrastructure: Application/Daemon settings classes</td>
<td>2477</td>
</tr>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td>mORMotDDD</td>
<td>Domain-Driven-Design toolbox for mORMot</td>
<td>2285</td>
</tr>
<tr>
<td>mORMotHttpClient</td>
<td>HTTP/1.1 RESTful JSON Client classes for mORMot</td>
<td>2306</td>
</tr>
<tr>
<td>mORMotHttpServer</td>
<td>HTTP/1.1 RESTFUL JSON Server classes for mORMot</td>
<td>2313</td>
</tr>
<tr>
<td>mORMotService</td>
<td>Daemon manageg classes for mORMot, including low-level Win NT Service</td>
<td>2370</td>
</tr>
<tr>
<td>SynBidirSock</td>
<td>Implements bidirectional client and server protocol, e.g. WebSockets</td>
<td>680</td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynCrtSock</td>
<td>Classes implementing TCP/UDP/HTTP client and server protocol</td>
<td>1083</td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
</tbody>
</table>
Synopse mORMot Framework
Software Architecture Design 1.18
Date: September 16, 2020

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynEcc</td>
<td>Certificate-based public-key cryptography using ECC-secp256r1 - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1313</td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1363</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/cache/search/multithread/OS features - as a complement to SynCommons, which tended to increase too much; licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1721</td>
</tr>
</tbody>
</table>

**dddInfraApps class hierarchy**

**Objects implemented in the dddInfraApps unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSynPersistentWithPassword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSynAutoCreateFields</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSQLRestThread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSQLHttpsClient</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSynAutoCreateFields</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSynPersistentWithPassword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TInterfacedObjectLocked</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TObject</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDThreadDaemon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAdministratedThreadDaemon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAdministratedRestDaemon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAdministratedDaemonMonitor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDSocketThreadMonitoring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDSocket</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDDDMockedSocket</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDDDSocket</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IInterface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EEDDMockedSocket</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDDDInfraException</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDDDException</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDDDRestClient</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dddInfraApps.pas unit - Rev. 1.18  Page 2452 of 2552
<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDDDMockedSocket</td>
<td>The default exception class raised by TDDDMockedSocket</td>
<td>2457</td>
</tr>
<tr>
<td>EDDDRestClient</td>
<td>Exception raised by TDDDRestClientSettings classes</td>
<td>2453</td>
</tr>
<tr>
<td>IDDDSocket</td>
<td>Interface allowing to customize/mock a socket connection</td>
<td>2455</td>
</tr>
<tr>
<td>TDDDMockedSocket</td>
<td>Implements IDDDSocket using a fake/mocked in-memory input/output storage</td>
<td>2457</td>
</tr>
<tr>
<td>TDDDRestClientDefiniti on</td>
<td>Advanced parameters for TDDDRestClientSettings definition</td>
<td>2453</td>
</tr>
<tr>
<td>TDDDRestClientHttp</td>
<td>Abstract client to connect to any daemon service via HTTP or HTTPS</td>
<td>2455</td>
</tr>
<tr>
<td>TDDDRestClientSettings</td>
<td>Storage class for initializing an ORM/SA REST Client class</td>
<td>2454</td>
</tr>
<tr>
<td>TDDDRestClientWebSockets</td>
<td>Abstract client to connect to any daemon service via WebSockets</td>
<td>2455</td>
</tr>
<tr>
<td>TDDDRestDaemon</td>
<td>Abstract class to implement a IAdministratedDaemon service via a TSQLRestServer</td>
<td>2453</td>
</tr>
<tr>
<td>TDDDRestHttpDaemon</td>
<td>Abstract class to implement a IAdministratedDaemon service via a TSQLRestServer, publishing its services as HTTP</td>
<td>2453</td>
</tr>
<tr>
<td>TDDDSocketThread</td>
<td>A generic TThread able to connect and reconnect to a TCP server</td>
<td>2459</td>
</tr>
<tr>
<td>TDDDSocketThreadMonitoring</td>
<td>The monitoring information of a TDDDSocketThread thread</td>
<td>2455</td>
</tr>
<tr>
<td>TDDDSynCrtSocket</td>
<td>Implements IDDDSocket using a SynCrtSock.TCrtSocket instance</td>
<td>2456</td>
</tr>
<tr>
<td>TDDDThreadDaemon</td>
<td>Abstract class to implement a IAdministratedDaemon service via a TThread</td>
<td>2452</td>
</tr>
</tbody>
</table>

**TDDDThreadDaemon = class(TDDDAdministratedThreadDaemon)**

Abstract class to implement a IAdministratedDaemon service via a TThread
- as hosted by TDDDDaemon service/daemon application

**procedure** SubscribeLog(const Levels: TSynLogInfos; const Callback: ISynLogCallback; ReceiveExistingKB: cardinal); **override**;

IAdministratedDaemon command to subscribe to a set of events for real-time remote monitoring of the specified log events
- this overridden method would disallow remote logs if low-level frames logging is set (i.e. HttpClientFullWebSocketsLog / HttpServerFullWebSocketsLog) to avoid an unexpected race condition

**property** AdministrationHTTPServer: TSQLHttpServer **read** GetAdministrationHTTPServer;

Reference to the HTTP server publishing IAdministratedDaemon service
- may equal nil if TDDDAdministratedDaemonSettingsFile.AuthHttp.BindPort=""
TDDDRestDaemon = class(TDDDAdministratedRestDaemon)

Abstract class to implement a IAdministratedDaemon service via a TSQLRestServer
- as hosted by TDDDDaemon service/daemon application

procedure SubscribeLog(const Levels: TSynLogInfos; const Callback: ISynLogCallback;
ReceiveExistingKB: cardinal); override;
IAdministratedDaemon command to subscribe to a set of events for real-time remote monitoring
of the specified log events
- this overridden method would disallow remote logs if low-level frames logging is set (i.e.
HttpClientFullWebSocketsLog / HttpServerFullWebSocketsLog) to avoid an unexpected race
condition

property AdministrationHTTPServer: TSQLHttpServer read
GetAdministrationHTTPServer;
Reference to the HTTP server publishing IAdministratedDaemon service
- may equal nil if TDDDDAdministratedDaemonSettingsFile.AuthHttp.BindPort="

TDDDRestHttpDaemon = class(TDDDRestDaemon)

Abstract class to implement a IAdministratedDaemon service via a TSQLRestServer, publishing its
services as HTTP
- as hosted by TDDDDaemon service/daemon application

procedure WrapperGenerate(const DestFile: TFileName; const Template: TFileName =
'API.adoc.mustache');
Generate API documentation corresponding to REST SOA interfaces

property HttpServer: TSQLHttpServer read fHttpServer;
Reference to the main HTTP server publishing this daemon Services
- may be nil outside a Start..Stop range

property ServicesLogRest: TSQLRest read fServicesLogRest;
Reference to the associated REST server storing the SOA log database
- may be nil if the daemon did not implement this feature

EDDDRestClient = class(EDDDException)

Exception raised by TDDDRestClientSettings classes

TDDDRestClientDefinition = class(TSynPersistentWithPassword)

Advanced parameters for TDDDRestClientSettings definition

property ConnectRetrySeconds: integer read fConnectRetrySeconds write
fConnectRetrySeconds;
How many seconds the client may try to connect after open socket failure

property Root: RawUTF8 read fRoot write fRoot;
The URI Root to be used for the REST Model

property WebSocketsPassword: RawUTF8 read fPassWord write fPassWord;
The encrypted password to be used to connect with WebSockets
TDDDRestClientSettings = class(TSynAutoCreateFields)

Storage class for initializing an ORM/SOA REST Client class
- this class will contain some generic properties to initialize a TSQLRestClientURI pointing to a remote server, using WebSockets by default
- WebSockets support is the reason why this class is defined in dddInfraApps, and not dddInfraSettings

function NewRestClientInstance(aRootSettings: TDDDAppSettingsAbstract; aModel: TSQLModel = nil; aOptions: TDDDNewRestInstanceOptions = [riOwnModel, riCreateVoidModelIfNone, riHandleAuthentication, riRaiseExceptionIfNoRest]): TSQLRestClientURI;

virtual;

is able to instantiate a Client REST instance for the stored definition
- Definition.Kind is expected to specify a TSQLRestClient class to be instantiated, not a TSQLRestServer instance
- will return nil if the supplied Definition is not correct
- note that the supplied Model.Root is expected to be the default root URI, which will be overriden with this TDDDRestSettings.Root property
- will also set the TSQLRest.LogFamily.Level from LogLevels value,

function OnAuthentificationFailed(Retry: integer; var aUserName, aPassword: string; out aPasswordHashed: boolean): boolean;

You may assign this method to a TSQLRestClientURI.OnAuthenticationFailed property, so that the client would automatically try to re-connect

procedure SetDefaults(const Root, Port, WebSocketsPassword, UserPassword: RawUTF8; const User: RawUTF8 = 'User'; const Server: RawUTF8 = 'localhost'; ForceSetCredentials: boolean = false; ConnectRetrySeconds: integer = 0; WebSocketsLoopDelayMS: integer = 0);

Set the default values for Client.Root, ORM.ServerName, Client.WebSocketsPassword and ORM.Password

property Client: TDDDRestClientDefinition read fClient;

Advanced connection options
- ORM.Password defines the authentication main password, and Client.WebSocketsPassword is used for WebSockets binary encryption

property ORM: TSynConnectionDefinition read fORM;

Defines a mean of access to a TSQLRest instance
- using Kind/ServerName/DatabaseName/User/Password properties: Kind would define the TSQLRest class to be instantiated by NewRestClientInstance()

property Timeout: integer read fTimeout write fTimeout;

You can overload here the TCP timeout delay, in seconds

property WebSocketsLoopDelay: integer read fWebSocketsLoopDelay write fWebSocketsLoopDelay;

You can overload here the WebSockets internal loop delay, in milliseconds
TDDDRestClientWebSockets = class(TSQLHttpClientWebsockets)
Abstract client to connect to any daemon service via WebSockets
- will monitor the connection, to allow automatic reconnection, with proper services resubscription

constructor Create(aSettings: TDDDRestClientSettings; aOnConnect: TOnRestClientNotify = nil; aOnDisconnect: TOnRestClientNotify = nil); reintroduce; overload; virtual;
Initialize the client instance with the supplied settings

destructor Destroy; override;
Finalize the client instance

property ApplicationName: RawUTF8 read fApplicationName;
Human-friendly application name, as set by overridden DefineApplication

property ApplicationVersion: RawUTF8 read GetSessionVersion;
Returns the server version, using timestamp/info method-based service

property Connected: boolean read fConnected;
Reflects the current WebSockets connection state

TDDDRestClientHttp = class(TSQLHttpsClient)
Abstract client to connect to any daemon service via HTTP or HTTPS
- defines a simple REST client, without connection tracking

property ApplicationName: RawUTF8 read fApplicationName;
Human-friendly application name, as set by overridden DefineApplication

property ApplicationVersion: RawUTF8 read GetSessionVersion;
Returns the server version, using timestamp/info method-based service

TDDDSocketThreadMonitoring = class(TDDDAdministratedDaemonMonitor)
The monitoring information of a TDDDSocketThread thread

property Owner: TObject read fOwner write fOwner;
May be a TDDDSocketThread instance, or not (to maintain a global state over several threads)

property Socket: variant read GetSocket;
Information about the associated socket

property State: TDDDSocketThreadState read fState write fState;
How this thread is currently connected to its associated TCP server

IDDDSocket = interface(IInterface)
Interface allowing to customize/mock a socket connection
function DataIn(Content: PAnsiChar; ContentLength: integer): integer;
    Get Length bytes from the (mocked) socket
    - returns the number of bytes read into the Content buffer
    - call e.g. TCrtSocket.SockInRead() method

function DataInPending(aTimeOut: integer): integer;
    Returns the number of bytes pending in the (mocked) socket
    - call e.g. TCrtSocket.SockInPending() with aSocketForceCheck=true, to return bytes both in the instance memory buffer and the socket API

function DataOut(Content: PAnsiChar; ContentLength: integer): boolean;
    Send Length bytes to the (mocked) socket
    - returns false on any error, true on success
    - call e.g. TCrtSocket.TrySndLow() method

function Handle: integer;
    Returns the low-level handle of this connection
    - is e.g. the socket file description

function Identifier: RawUTF8;
    Returns instance identifier
    - e.g. the TCrtSocket.Sock number as text

function LastError: RawUTF8;
    Get some low-level information about the last occurred error
    - e.g. TCrtSocket.LastLowSocketError value

procedure Connect;
    Connect to the host via the (mocked) socket
    - should raise an exception on error
function DataOut(Content: PAnsiChar; ContentLength: integer): boolean;
Call TCrtSocket.TrySndLow() method

function Handle: integer;
Returns TCrtSocket.Sock

function Identifier: RawUTF8;
Returns TCrtSocket.Sock number as text

function LastError: RawUTF8;
Get information from TCrtSocket.LastLowSocketError

procedure Connect;
Call TCrtSocket.OpenBind

property Owner: TThread read fOwner;
Read-only access to the associated processing thread
- not published, to avoid stack overflow since TDDDSocketThreadMonitoring would point to this instance

property Socket: TCrtSocket read fSocket;
Read-only access to the associated processing socket

EDDDMockedSocket = class(EDDDInfraException)
The default exception class raised by TDDDMockedSocket

TDDDMockedSocket = class(TInterfacedObjectLocked)
Implements IDDDSocket using a fake/mocked in-memory input/output storage
- may be supplied to TDDDSocketThread to bypass its default network communication
- you could fake input/output of TCP/IP packets by calling MockDataIn() and MockDataOut() methods - incoming and outgoing packets would be merged in the internal in-memory buffers, as with a regular Socket
- you could fake exception, for any upcoming method call, via MockException()
- you could emulate network latency, for any upcoming method call, via MockLatency() - to emulate remote/wireless access, or thread pool contention
- this implementation is thread-safe, so multiple threads could access the same IDDDSocket instance, and settings be changed in real time

constructor Create(aOwner: TThread); reintroduce; virtual;
Initialize the mocked socket instance

function DataIn(Content: PAnsiChar; ContentLength: integer): integer;
IDDDSocket method to get Length bytes from the mocked socket
- returns the number of bytes read into the Content buffer
- note that all pending data is returned as once, i.e. all previous calls to MockDataIn() would be gathered in a single buffer
function DataInPending(aTimeOut: integer): integer;

IDDDSocket method to return the number of bytes pending
- note that the total length of all pending data is returned as once, i.e. all previous calls to
MockDataIn() would be sum as a single count
- this method will emulate blocking process, just like a regular socket: if there is no pending data,
it will wait up to aTimeOut milliseconds

function DataOut(Content: PAnsiChar; ContentLength: integer): boolean;

IDDDSocket method to send Length bytes to the mocked socket
- returns false on any error, true on success
- then MockDataOut could be used to retrieve the sent data

function Handle: integer;
Returns 0 (no associated file descriptor)

function Identifier: RawUTF8;
IDDDSocket method to return a fake instance identifier
- in fact, the hexa pointer of the TDDDMockedSocket instance

function LastError: RawUTF8;
IDDDSocket method to get some low-level information about the last error
- i.e. the latest ExceptionMessage value as set by MockException()

function MockDataOut: RawByteString;
Return the bytes from the internal fake output storage
- as has be previously set by the DataOut() method
- will gather all data from several DataOut() calls in a single buffer

procedure Connect;
IDDDSocket method to connect to the host via the mocked socket
- won't raise any exception unless ConnectShouldCheckRaiseException is set

procedure MockDataIn(const Content: RawByteString);
Add some bytes to the internal fake input storage
- would be made accessible to the DataInPending/DataIn methods
- the supplied buffer would be gathered to any previous MockDataIn() call, which has not been
read yet by the DataIn() method
procedure MockException(NextActions: TDDDMockedSocketExceptions; const ExceptionMessage: string = ''; ExceptionClass: ExceptClass = nil);

- The specified methods would raise an exception
- only a single registration is memorized: once raised, any further method execution would continue as usual
- optional Exception.Message which should be raised with the exception
- also optional exception class instead of default EDDDMockedSocket
- msaDataOutReturnsFalse won't raise any exception, but let DataOut method return false (which is the normal way of indicating a socket error) - in this case, ExceptionMessage would be available from LastError
- msaDataInPendingTimeout won't raise any exception, but let DataInPending sleep for the timeout period, and return 0
- msaDataInPendingFails won't raise any exception, but let DataInPending fails immediately, and return -1 (emulating a broken socket)
- you may use ALL_DDDMOCKED_EXCEPTIONS to set all possible actions
- you could reset any previous registered exception by calling MockException([1]);

procedure MockLatency(NextActions: TDDDMockedSocketLatencies; MilliSeconds: integer);

- Will let the specified methods to wait for a given number of milliseconds
- allow to emulate network latency, on purpose
- you may use ALL_DDDMOCKED_LATENCIES to slow down all possible actions

property Owner: TThread read fOwner;

- Read-only access to the associated processing thread
- not published, to avoid stack overflow since TDDDSocketThreadMonitoring would point to this instance

property PendingInBytes: integer read GetPendingInBytes;

- How many bytes are actually in the internal input buffer

property PendingOutBytes: integer read GetPendingOutBytes;

- How many bytes are actually in the internal output buffer

TDDDSocketThread = class(TSQLRestThread)

- A generic TThread able to connect and reconnect to a TCP server
- initialize and own a TCrtSocket instance for TCP transmission
- allow automatic reconnection
- inherit from TSQLRestThread, so should be associated with a REST instance

constructor Create(aSettings: TDDDSocketThreadSettings; aRest: TSQLRest; aMonitoring: TDDDSocketThreadMonitoring);

- Initialize the thread for a given REST instance

destructor Destroy; override;

- Finalize the thread process, and its associated REST instance
```pascal
function StatsAsJson: RawUTF8;
  Returns the Monitoring and Rest statistics as a JSON object
  - resulting format is
    {...MonitoringProperties...,"Rest":{...RestStats...}}

procedure Shutdown(andTerminate: boolean); virtual;
  Will pause any communication with the associated socket
  - could be used before stopping the service for cleaner shutdown

property Host: SockString read fHost;
  The IP Host name used to connect with TCP

property Port: SockString read fPort;
  The IP Port value used to connect with TCP

property Settings: TDDDSocketThreadSettings read fSettings;
  The parameters used to setup this thread process

Types implemented in the dddInfraApps unit

TDDDMockedSocketException = ( msaConnectRaiseException, msaDataInPendingTimeout, msaDataInPendingFails, msaDataInRaiseException, msaDataOutRaiseException, msaDataOutReturnsFalse );
  Defines the potential mocked actions for TDDDMockedSocket.MockException()

TDDDMockedSocketExceptions = set of TDDDMockedSocketException;
  Defines a set of mocked actions for TDDDMockedSocket.MockException()

TDDDMockedSocketLatencies = set of TDDDMockedSocketLatency;
  Defines a set of mocked actions for TDDDMockedSocket.MockLatency()

TDDDMockedSocketLatency = ( mslConnect, mslDataIn, mslDataOut );
  Defines the potential mocked actions for TDDDMockedSocket.MockLatency()

TDDDSocketThreadState = ( tpsDisconnected, tpsConnecting, tpsConnected );
  The current connection state of the TCP client associated to a TDDDSocketThread thread

TECCAuthorize = ( eaSuccess, eaInvalidSecret, eaMissingUnlockFile, eaInvalidUnlockFile, eaInvalidJson );
  Result codes of the ECCAuthorize() function

Constants implemented in the dddInfraApps unit

ALL_DDMOCKED_EXCEPTIONS = [msaConnectRaiseException, msaDataInPendingFails, msaDataInRaiseException, msaDataOutReturnsFalse];
  Map realistic exceptions steps for a mocked socket
  - could be used to simulate a global socket connection drop

ALL_DDMOCKED_LATENCIES = [Low(TDDDMockedSocketLatency)..high(TDDDMockedSocketLatency)];
  Map realistic latencies steps for a mocked socket
  - could be used to simulate a slow network

EXEVERSION_RCTEMPLATE: RawUTF8 = '1 VERSIONINFO'##10 + 'FILEVERSION
```
A Mustache template of a .rc version information
- could be used to compile a custom .res version file in an automated way
- if you use this generated .res, ensure your "Version Info" is disabled (unchecked) in the Delphi IDE project options

Functions or procedures implemented in the `dddInfraApps` unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AdministratedDaemonClient</code></td>
<td>Create a client safe asynchronous connection to a <code>IAdministratedDaemon</code> service</td>
<td>2461</td>
</tr>
<tr>
<td><code>AdministratedDaemonServer</code></td>
<td>Create a WebSockets server instance, publishing a <code>IAdministratedDaemon</code> service</td>
<td>2461</td>
</tr>
<tr>
<td><code>ECCAuthorize</code></td>
<td>Any sensitive, or licensed program, could call this method to check for authorized execution for a given user on a given computer, using very secure asymmetric ECC cryptography</td>
<td>2462</td>
</tr>
</tbody>
</table>

```delphi
function AdministratedDaemonClient(Definition: TDDDRestClientSettings; Model: TSQLModel = nil): TSQLHttpClientWebsockets;
  Create a client safe asynchronous connection to a `IAdministratedDaemon` service

function AdministratedDaemonServer(Settings: TDDDAdministratedDaemonSettings; DaemonClass: TDDDAdministratedDaemonClass): TDDDAdministratedDaemon;
  Create a WebSockets server instance, publishing a `IAdministratedDaemon` service
```
function ECCAuthorize(aContent: TObject; aSecretDays: integer; const aSecretPass, aDPAPI, aDecryptSalt, aAppLockPublic64: RawUTF8; const aSearchFolder: TFileName = ''; aSecretInfo: PECCCertificateSigned = nil; aLocalFile: PFileName = nil): TECCAuthorize;

Any sensitive, or licensed program, could call this method to check for authorized execution for a given user on a given computer, using very secure asymmetric ECC cryptography

- applock.public/private keys pair should have been generated, applock.public stored as aAppLockPublic64 in the executables, and applock.private kept secret
- will search for encrypted authorization in a local user@host.unlock file
- if no user@host.unlock file is found, will create local user@host.public and user@host.secret files and return eaMissingUnlockFile: user should then send user@host.public to the product support to receive its user@host.unlock file (a dedicated UI may be developed, or an uncrypted email can be used for transfer with the support team, thanks to asymmetric cryptography)
- local user@host.secret file is encrypted via DPAPI/CryptDataForCurrentUser for the specific computer and user (to avoid .unlock reuse on another PC)
- support team should create a user@host.json file matching aContent: TObject published properties, containing all application-specific settings and authorization scope; then it could create the unlock file using e.g. an unlock.bat file running the ECC tool over secret applock.private keys:

```
@echo off
echo Usage:  unlock user@host
echo.
ecc sign -file %1.json -auth applock -pass applockprivatepassword -rounds 60000
ecc crypt -file %1.json -out %1.unlock -auth %1 -saltpass decryptsalt -saltrounds 10000
del %1.json.sign
```

- returns eaInvalidUnlockFile if the local user@host.unlock file is not correctly signed and encrypted for this user (e.g. corrupted or deprecated)
- eaInvalidJson will indicate some error in the .json created by support team, i.e. if it does not match aContent: TObject published properties
- eaSuccess should let the application execute, on the returned scope
- returns eaSuccess if a local user@host.unlock file has been successfully decrypted and validated (using ECDSA over aAppLockPublic64) and successfully unserialized from JSON into aContent object instance
- user@host.* files are searched in the executable folder if aSearchFolder='', but you may specify a custom location, e.g. use ECCKeyFileFolder
- aSecretPass could be entered by the end-user, to authenticate its identity; you may specify a string constant if local applock.public/private key files is enough secure for your application
- will use the supplied aDPAPI/aDecryptSalt parameters to restrict this authorization to a specific product (i.e. isolate the execution context to reduce forensic scope), for dedicated applock.public/private keys pair - just pass some application-specific string constant to those parameters
- aSecretInfo^ could be set to retrieve the user@host.secret information (e.g. validity dates), and aLocalFile^ the '<fullpath>user@host' file prefix

Variables implemented in the dddInfraApps unit

GlobalCopyright:  string = '';

You could set a text to this global variable at runtime, so that it would be displayed as copyright older name for the console
27.81. dddInfraAuthRest.pas unit

*Purpose:* Shared DDD Infrastructure: Authentication implementation
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the dddInfraAuthRest unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dddDomAuthInterfaces</td>
<td>Shared DDD Domains: Authentication objects and interfaces</td>
<td>2437</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMotDDD</td>
<td>Domain-Driven-Design toolbox for mORMot</td>
<td>2285</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
<tr>
<td></td>
<td>- implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JWT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- optimized for speed (tuned assembler and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SSE3/SSE4/AES-NI/PADLOCK support)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTests</td>
<td>Unit test functions used by Synopse projects</td>
<td>1832</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

**Objects implemented in the dddInfraAuthRest unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLRecord</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSQLRecordUserAuth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationAbstract</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationMD5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationSHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationSHA384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationSHA512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationSHA3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationSHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationSHA512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationSHA3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationRestFactoryMD5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationRestFactorySHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationRestFactorySHA384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationRestFactorySHA512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationRestFactorySHA3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**dddInfraAuthRest class hierarchy**

**Objects implemented in the dddInfraAuthRest unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDDDRepositoryRestCommand</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationAbstract</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationMD5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationSHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationSHA384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationSHA512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationSHA3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationRestFactoryMD5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationRestFactorySHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationRestFactorySHA384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationRestFactorySHA512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDDDAuthenticationRestFactorySHA3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ddInfraAuthRest class hierarchy**
<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDDDAuthenticationAbstract</td>
<td>Generic class for implementing authentication</td>
<td>2464</td>
</tr>
<tr>
<td>TDDDAuthenticationMD5</td>
<td>Implements authentication using MD5 hashing</td>
<td>2465</td>
</tr>
<tr>
<td>TDDDAuthenticationResFactoryAbstract</td>
<td>Abstract factory of IDomAuthCommand repository instances using REST</td>
<td>2465</td>
</tr>
<tr>
<td>TDDDAuthenticationResFactoryMD5</td>
<td>Factory of IDomAuthCommand repository instances using a RESTful ORM access and SHA-256 hashing algorithm</td>
<td>2466</td>
</tr>
<tr>
<td>TDDDAuthenticationResFactorySHA256</td>
<td>Factory of IDomAuthCommand repository instances using a RESTful ORM access and SHA-256 hashing algorithm</td>
<td>2465</td>
</tr>
<tr>
<td>TDDDAuthenticationSHA256</td>
<td>Implements authentication using SHA-256 hashing</td>
<td>2465</td>
</tr>
<tr>
<td>TSQLRecordUserAuth</td>
<td>ORM object to persist authentication information, i.e. TAuthInfo</td>
<td>2464</td>
</tr>
</tbody>
</table>

TSQLRecordUserAuth = class(TSQLRecord)

ORM object to persist authentication information, i.e. TAuthInfo

**property** HashedPassword: RawUTF8 **read** fHashedPassword **write** fHashedPassword;

The password, stored in a hashed form
- this property does not exist at TAuthInfo level, so will be private to the storage layer - which is the safest option possible

**property** Logon: RawUTF8 **read** fLogon **write** fLogon stored AS_UNIQUE;

Will map TAuthInfo.LogonName
- is defined as "stored AS_UNIQUE" so that it may be used as primary key

TDDDAuthenticationAbstract = class(TDDDRepositoryRestCommand)

Generic class for implementing authentication
- do not instantiate this abstract class, but e.g. TDDDAuthenticationSHA256 or TDDDAuthenticationMD5

**function** Add(const aLogonName: RawUTF8; aHashedPassword: TAuthQueryNonce): TCQRSResult;

Register a new credential, from its LogonName/HashedPassword values
- on success, the newly created credential will be the currently selected

**function** ChallengeSelectFinal(const aChallengedPassword: TAuthQueryNonce): TCQRSResult;

Validate the first phase of a dual pass challenge authentication

**function** ChallengeSelectFirst(const aLogonName: RawUTF8): TAuthQueryNonce;

Initiate the first phase of a dual pass challenge authentication
class function ClientComputeChallengedPassword( const aLogonName,aPlainPassword: RawUTF8; const aChallengeFromServer: TAuthQueryNonce): TAuthQueryNonce; virtual;
    Class method to be used on the client side to resolve the challenge
    - is basically
      result := DoHash(aLogonName+':'+aChallengeFromServer+':'+
                      ComputeHashPassword(aLogonName,aPlainPassword));

class function ComputeHashPassword(const aLogonName,aPassword: RawUTF8): TAuthQueryNonce;
    Class method to be used to compute a password hash from its plain value

function Get(out aAggregate: TAuthInfo): TCQRSResult;
    Retrieve some information about the current selected credential

function Logged: boolean;
    Returns TRUE if the dual pass challenge did succeed

function LogonName: RawUTF8;
    Returns the logon name of the authenticated user

function SelectByName(const aLogonName: RawUTF8): TCQRSResult;
    Set the credential for Get() or further IDomAuthCommand.Update/Delete
    - this method execution will be disabled for most clients

function UpdatePassword(const aHashedPassword: TAuthQueryNonce): TCQRSResult;
    Update the current selected credential password

class procedure RegressionTests(test: TSynTestCase);
    Built-in simple unit tests

TDDDAuthenticationSHA256 = class(TDDDAuthenticationAbstract)
    Implements authentication using SHA-256 hashing
    - more secure than TDDDAuthenticationMD5

TDDDAuthenticationMD5 = class(TDDDAuthenticationAbstract)
    Implements authentication using MD5 hashing
    - less secure than TDDDAuthenticationSHA256

TDDDAuthenticationRestFactoryAbstract = class(TDDDRepositoryRestFactory)
    Abstract factory of IDomAuthCommand repository instances using REST
    constructor Create(aRest: TSQLRest; aImplementationClass: TDDDAuthenticationClass; aOwner: TDDDRepositoryRestManager); reintroduce;
            Initialize a factory with the supplied implementation algorithm

TDDDAuthenticationRestFactorySHA256 = class(TDDDAuthenticationRestFactoryAbstract)
    Factory of IDomAuthCommand repository instances using a RESTful ORM access and SHA-256 hashing algorithm
constructor Create(aRest: TSQLRest; aOwner: TDDDRepositoryRestManager=nil);
reintroduce;

*Initialize a factory with the SHA-256 implementation algorithm*

```
TDDDAuthenticationRestFactoryMD5 =
class(TDDDAuthenticationRestFactoryAbstract)
  Factory of IDomAuthCommand repository instances using a RESTful ORM access and SHA-256 hashing algorithm
  constructor Create(aRest: TSQLRest; aOwner: TDDDRepositoryRestManager=nil);
  reintroduce;
  *Initialize a factory with the SHA-256 implementation algorithm*
```

Types implemented in the *dddInfraAuthRest* unit

```
TDDDAuthenticationClass = class of TDDDAuthenticationAbstract;
  Allows to specify which actual hashing algorithm would be used
  - i.e. either TDDDAuthenticationSHA256 or TDDDAuthenticationMD5
```
27.82. dddInfraEmail.pas unit

**Purpose:** Shared DDD Infrastructure: implement an email validation service
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

### Units used in the dddInfraEmail unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| dddDomUserInterfaces | Shared DDD Domains: User interfaces definition  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2445 |
| dddDomUserTypes   | Shared DDD Domains: User objects definition  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2447 |
| mORMot            | Common ORM and SOA classes for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1899 |
| mORMotDDD         | Domain-Driven-Design toolbox for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 2285 |
| SynCommons        | Common functions used by most Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 717  |
| SynCrypto         | Fast cryptographic routines (hashing and cypher)  
- implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT  
- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1139 |
| SynTable          | Filter/database/cache/buffer/security/search/multithread/OS features as a complement to SynCommons, which tended to increase too much  
- licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1721 |
| SynTests          | Unit test functions used by Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1832 |
Objects implemented in the **dddInfraEmail** unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDDDEmail</td>
<td>Exception raised during any email process of this DDD's infrastructure implementation</td>
<td>2468</td>
</tr>
<tr>
<td>TDDDEmailRedirection</td>
<td>Parameters used for the validation link of an email address</td>
<td>2468</td>
</tr>
<tr>
<td>TDDDEmailServiceAbstract</td>
<td>Abstract parent of any email-related service</td>
<td>2469</td>
</tr>
<tr>
<td>TDDDEmailValidation</td>
<td>Parameters used for the validation/verification process of an email address</td>
<td>2469</td>
</tr>
<tr>
<td>TDDDEmailValidationService</td>
<td>Service used to validate an email address via an URL link to be clicked</td>
<td>2469</td>
</tr>
<tr>
<td>TSQLRecordEmailAbstract</td>
<td>ORM class storing an email in addition to creation/modification timestamps</td>
<td>2470</td>
</tr>
<tr>
<td>TSQLRecordEmailValidation</td>
<td>ORM class for email validation process</td>
<td>2470</td>
</tr>
</tbody>
</table>

**EDDDEmail = class(EDDDInfraException)**

*Exception raised during any email process of this DDD's infrastructure implementation*

**TDDDEmailRedirection = class(TSynPersistent)**

*Parameters used for the validation link of an email address*
- may be stored as daemon/service level settings, using e.g. dddInfraSettings

**property** RestServerPublicRootURI: RawUTF8 read fRestServerPublicRootURI write fRestServerPublicRootURI;

*The public URI which would be accessible from the Internet*
- may be e.g 'http://publicserver/restroot'
property SuccessRedirectURI: RawUTF8 read fSuccessRedirectURI write fSuccessRedirectURI;
The URI on which the browser will be redirected on validation success
- you can specify some '%' parameter markers, ordered as logon, email, and validation IP
- may be e.g. 'http://publicwebsite/success&logon=%'

property ValidationMethodName: RawUTF8 read fValidationMethodName write fValidationMethodName;
The validation method name for the URI
- if not set, TDDDEmailValidationService will use 'EmailValidate'
- clickable URI would be RestServerPublicRootURI+/+ValidationMethodName

TDDDEmailValidation = class(TSynAutoCreateFields)
Parameters used for the validation/verification process of an email address
- may be stored as daemon/service level settings, using e.g. dddInfraSettings

procedure SetDefaultValuesIfVoid(const aSenderEmail, aApplication, aRedirectionURIPublicRoot, aRedirectionURISuccess: RawUTF8);
Will fill some default values in the properties, if none is set

property Redirection: TDDDEmailRedirection read fRedirection;
Parameters defining the validation link of an email address

property Template: TDomUserEmailTemplate read fTemplate;
How the email should be created from a given template

property TemplateFolder: TFileName read fTemplateFolder write fTemplateFolder;
Where the template files are to be found

TDDDEmailServiceAbstract = class(TCQRSQueryObjectRest)
Abstract parent of any email-related service
- will define some common methods to validate an email address

property EmailValidate: TSynValidate read fEmailValidate write SetEmailValidate;
Direct access to the email validation instance
- you can customize the default TSynValidateEmail to meet your own expectations - once set, it will be owned by this class instance

TDDDEmailValidationService = class(TDDDEmailServiceAbstract)
Service used to validate an email address via an URL link to be clicked

constructor Create(aRest: TSQLRest); override;
Initialize the validation service for a given ORM persistence
- would recognize the TSQLRecordEmailValidation class from aRest.Model
- will use aRest.Services for IoC, e.g. EMailer/Template properties

function ComputeURIForReply(const aLogonName, aEmail: RawUTF8): RawUTF8;
Compute the target URI corresponding to SetURIForServer() parameters
function IsEmailValidated(const aLogonName, aEmail: RawUTF8): boolean; virtual;
    Check if an email has been validated for a given logon

function StartEmailValidation(const aTemplate: TDomUserEmailTemplate; const aLogonName, aEmail: RawUTF8): TCQRSResult; virtual;
    Check the supplied parameters, and send an email for validation

procedure SetURIForServer(aRestServerPublic: TSQLRestServer; aParams: TDDDEmailRedirection); overload;
    Register the callback URI service

procedure SetURIForServer(aRestServerPublic: TSQLRestServer; const aRestServerPublicRootURI, aSuccessRedirectURI, aValidationMethodName: RawUTF8); overload;
    Register the callback URI service
    - same as the overloaded function, but taking parameters one by one

property EMailer: IDomUserEmailer read fEmailer;
    Will be injected (and freed) with the emailer service

property RestClass: TSQLRecordEmailValidationClass read fRestClass;
    The associated ORM class used to store the email validation process
    - any class inheriting from TSQLRecordEmailValidation in the aRest.Model will be recognized by Create(aRest) to store its information
    - this temporary storage should not be the main user persistence domain

property Template: IDomUserTemplate read fTemplate;
    Will be injected (and freed) with the email template service

property ValidationServerRoot: RawUTF8 read fValidationServerRoot;
    The public URI which would be accessible from the Internet
    - may be e.g. 'http://publicserver/restroot'

property ValidationURI: RawUTF8 read fValidationMethodName;
    The validation method name for the URI
    - if not set, TDDDEmailValidationService will use 'EmailValidate'
    - clickable URI would be ValidationServerRoot+'/'+ValidationMethodName

TSQLRecordEmailAbstract = class(TSQLRecordTimed)
    ORM class storing an email in addition to creation/modification timestamps
    - declared as its own class, since may be reused

property Email: RawUTF8 read fEmail write fEmail;
    The stored email address

TSQLRecordEmailValidation = class(TSQLRecordEmailAbstract)
    ORM class for email validation process
    - we do not create a whole domain here, just an ORM persistence layer
    - any class inheriting from TSQLRecordEmailValidation in the Rest.Model will be recognized by TDDDEmailValidationService to store its information
27.83. dddInfraEmailer.pas unit

**Purpose:** Shared DDD Infrastructure: generic emailing service
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the dddInfraEmailer unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dddDomUserInterfaces</td>
<td>Shared DDD Domains: User interfaces definition</td>
<td>2445</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>dddDomUserTypes</td>
<td>Shared DDD Domains: User objects definition</td>
<td>2447</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>dddInfraEmail</td>
<td>Shared DDD Infrastructure: implement an email validation service</td>
<td>2467</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMotDDD</td>
<td>Domain-Driven-Design toolbox for mORMot</td>
<td>2285</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrtSock</td>
<td>Classes implementing TCP/UDP/HTTP client and server protocol</td>
<td>1083</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynMustache</td>
<td>Logic-less mustache template rendering</td>
<td>1450</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynTests</td>
<td>Unit test functions used by Synopse projects</td>
<td>1832</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>
Objects implemented in the *dddInfraEmailer* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISMTPServerConnection</td>
<td>Used to inject the exact SMTP process to TDDDEmailerDaemon</td>
<td>2472</td>
</tr>
<tr>
<td>TDDDEmailerDaemon</td>
<td>Daemon used to send emails via SMTP</td>
<td>2473</td>
</tr>
<tr>
<td>TDDDEmailerDaemonProcess</td>
<td>Thread processing a SMTP connection</td>
<td>2473</td>
</tr>
<tr>
<td>TDDDEmailerDaemonStats</td>
<td>Statistics about a TDDDEmailerDaemon instance</td>
<td>2473</td>
</tr>
<tr>
<td>TDDDTemplateAbstract</td>
<td>Abstract Mustache-Based templating</td>
<td>2474</td>
</tr>
<tr>
<td>TDDDTemplateFromFolder</td>
<td>Mustache-Based templating from a local folder</td>
<td>2474</td>
</tr>
<tr>
<td>TSMTPServer</td>
<td>Abstract class used to resolve ISMTPServerConnection</td>
<td>2473</td>
</tr>
<tr>
<td>TSMTPServerSocketConnection</td>
<td>Implements ISMTPServerConnection using SynCrtSock's low-level SMTP access</td>
<td>2473</td>
</tr>
<tr>
<td>TSMTPServerSocketConnectionAbstract</td>
<td>Implements an abstract ISMTPServerConnection class</td>
<td>2473</td>
</tr>
<tr>
<td>TSQLRecordEmailer</td>
<td>ORM class for email validation process</td>
<td>2474</td>
</tr>
</tbody>
</table>

**ISMTPServerConnection** = `interface(IInvokable)`

*Used to inject the exact SMTP process to TDDDEmailerDaemon*

```
function SendEmail(const aRecipient: TRawUTF8DynArray; const aSender,aSubject,aHeader,aBody: RawUTF8): RawUTF8;
```

*This method should send the email, returning an error message on issue - if no header is supplied, it will expect one UTF-8 encoded text message*
TSMTPTServer = class(TInterfaceResolverForSingleInterface)

Abstract class used to resolve ISMTPServerConnection
- see TSMTPTServerSocket for actual implementation

constructor Create(aImplementation: TInterfacedObjectClass; aParameters: TSMTPTServer); overload;

Initialize the class with the parameters of another TSMTPTServer instance
- in fact, TSMTPTServer could be used as parameter storage of its needed published properties,
  e.g. in a TApplicationSettingsAbstract sub-class

constructor Create(aImplementation: TInterfacedObjectClass; const aAddress: RawUTF8; aPort: cardinal; const aLogin, aPassword: RawUTF8); overload;

Initialize the class with the supplied parameters

procedure SetDefaultValuesIfVoid;

Will fill some default values in the properties, if none is set
- i.e. 'dummy:dummy@localhost:25'

TSMTPTServerSocketConnectionAbstract = class(TInterfacedObject)

Implements an abstract ISMTPServerConnection class

TSMTPTServerSocketConnection = class(TSMTPServerSocketConnectionAbstract)

Implements ISMTPServerConnection using SynCrtSock's low-level SMTP access

TDDDEmailerDaemonStats = class(TSynMonitorWithSize)

Statistics about a TDDDEmailerDaemon instance
- in addition to a standard TSynMonitor, will maintain the connection count

procedure NewConnection;

Will increase the connection count

property Connection: cardinal read fConnection;

The connection count

TDDDEmailerDaemonProcess = class(TDDDMonitoredDaemonProcessRest)

Thread processing a SMTP connection

TDDDEmailerDaemon = class(TDDDMonitoredDaemon)

Daemon used to send emails via SMTP
- it will maintain a list of action in a TSQLRecordEmailer ORM storage

function SendEmail(const aRecipients: TRawUTF8DynArray; const aSender, aSubject, aHeaders, aBody: RawUTF8): TCQRSResult;

This is the main entry point of this service
- here the supplied message body is already fully encoded, as expected by SMTP (i.e. as one text
  message, or multi-part encoded)
- if no header is supplied, it will expect one UTF-8 encoded text message
property RestClass: TSQLRecordEmailerClass read fRestClass;

The associated class TSQLRecordEmailer used for status persistence
- any class inheriting from TSQLRecordEmailer in the Rest.Model will be recognized by
TDDDEmailerDaemon to store its information

property SMTPServer: TSMTPServer read fSMTPServer write fSMTPServer;

The associated class used as actual SMTP client

TSQLRecordEmailer = class(TSQLRecordTimed)
ORM class for email validation process
- we do not create a whole domain here, just an ORM persistence layer

class procedure InitializeTable(Server: TSQLRestServer; const FieldName: RawUTF8;
Options: TSQLInitializeTableOptions); override;
Will create an index on State+ID

TDDDDTemplateAbstract = class(TCQRSService)
Abstract Mustache-Based templating

TDDDDTemplateFromFolder = class(TDDDDTemplateAbstract)
Mustache-Based templating from a local folder

Types implemented in the dddInfraEmailer unit
TSQLRecordEmailerState = ( esPending, esSending, esSent, esFailed );
State machine used during email validation process

Functions or procedures implemented in the dddInfraEmailer unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestDddInfraEmailer</td>
<td>You can call this function within a TSynTestCase class to validate the email validation via a full regression set</td>
<td>2474</td>
</tr>
</tbody>
</table>

procedure TestDddInfraEmailer(serverClass: TSQLRestServerClass; test: TSynTestCase);

You can call this function within a TSynTestCase class to validate the email validation via a full regression set
- could be used as such:

procedure TTestCrossCuttingFeatures.Emailer;
begin // TSQLRestServerDB is injected to avoid any dependency to mORMotSQLite3
TestDddInfraEmailer(TSQLRestServerDB, self);
end;
27.84. dddInfraRepoUser.pas unit

**Purpose**: Shared DDD Infrastructure: User CQRS Repository via ORM
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the dddInfraRepoUser unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dddDomUserCQRS</strong></td>
<td>Shared DDD Domains: User CQRS Repository interfaces</td>
<td>2442</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><strong>dddDomUserTypes</strong></td>
<td>Shared DDD Domains: User objects definition</td>
<td>2447</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><strong>mORMot</strong></td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><strong>mORMotDDD</strong></td>
<td>Domain-Driven-Design toolbox for mORMot</td>
<td>2285</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><strong>SynCommons</strong></td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><strong>SynCrypto</strong></td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
<tr>
<td></td>
<td>- implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><strong>SynTable</strong></td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><strong>SynTests</strong></td>
<td>Unit test functions used by Synopse projects</td>
<td>1832</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

*ddInfraRepoUser class hierarchy*
Objects implemented in the `dddInfraRepoUser` unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TInfraRepoUser</td>
<td>Implements a User CQRS Repository via mORMot's RESTful ORM</td>
<td>2476</td>
</tr>
<tr>
<td>TInfraRepoUserFactory</td>
<td>Implements a Factory of User CQRS Repositories via mORMot's RESTful ORM</td>
<td>2476</td>
</tr>
<tr>
<td>TSQLRecordPerson</td>
<td>ORM class able to store a TPerson object</td>
<td>2476</td>
</tr>
<tr>
<td>TSQLRecordPersonContactable</td>
<td>ORM class able to store a TPersonContactable object</td>
<td>2476</td>
</tr>
<tr>
<td>TSQLRecordUser</td>
<td>ORM class used to persist a TUser domain aggregate</td>
<td>2476</td>
</tr>
</tbody>
</table>

TInfraRepoUser = `class` (TDDDRepositoryRestCommand)

*Implements a User CQRS Repository via mORMot's RESTful ORM*
- this class will use a supplied TSQLRest instance to persist TUser Aggregate Roots, following the IDomUserCommand CQRS methods
- each TUser aggregate will be mapped into a TSQLRecordUser ORM table

TInfraRepoUserFactory = `class` (TDDDRepositoryRestFactory)

*Implements a Factory of User CQRS Repositories via mORMot's RESTful ORM*
- this class will associate the TUser Aggregate Root with a TSQLRecordUser ORM table, as managed in a given TSQLRest instance

**constructor** Create(aRest: TSQLRest; aOwner: TDDDRepositoryRestManager=nil);
**reintroduce**

- Initialize the association with the ORM

**class procedure** RegressionTests(test: TSynTestCase);

- Perform some tests on this Factory/Repository implementation

TSQLRecordPerson = `class` (TSQLRecord)

*ORM class able to store a TPerson object*
- the TPerson.Name property has been flattened to Name_* columns as expected by TDDDRepositoryRestFactory.ComputeMapping

TSQLRecordPersonContactable = `class` (TSQLRecordPerson)

*ORM class able to store a TPersonContactable object*
- the TPersonContactable.Address property has been flattened to Address_* columns as expected by TDDDRepositoryRestFactory.ComputeMapping

TSQLRecordUser = `class` (TSQLRecordPersonContactable)

*ORM class used to persist a TUser domain aggregate*
27.85. dddInfraSettings.pas unit

Purpose: Shared DDD Infrastructure: Application/Daemon settings classes
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the dddInfraSettings unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMotDB</td>
<td>Virtual Tables for external DB access for mORMot</td>
<td>2276</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMotDDD</td>
<td>Domain-Driven-Design toolbox for mORMot</td>
<td>2285</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMotMongoDB</td>
<td>Direct optimized MongoDB access for mORMot's ORM</td>
<td>2334</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMotSQLite3</td>
<td>SQLite3 embedded Database engine used as the mORMot SQL kernel</td>
<td>2382</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMotWrappers</td>
<td>Generate cross-platform clients code and documentation from a mORMot server</td>
<td>2430</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrtSock</td>
<td>Classes implementing TCP/UDP/HTTP client and server protocol</td>
<td>1083</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
<tr>
<td></td>
<td>- implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)</td>
<td></td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>
**Unit Name** | **Description** | **Page**
---|---|---
**SynLog** | Logging functions used by Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1363

**SynMongoDB** | MongoDB document-oriented database direct access classes  
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1396

**SynSQLite3** | SQLite3 Database engine direct access  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1646

**SynTable** | Filter/database/cache/buffer/security/search/multithread/OS features  
- as a complement to SynCommons, which tended to increase too much  
- licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1721

---

### Objects implemented in the `dddInfraSettings` unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TDDDAdministratedDaemonHttpSettings</code></td>
<td>Parent class for storing a HTTP published service/daemon settings</td>
<td>2487</td>
</tr>
<tr>
<td><code>TDDDAdministratedDaemonRemoteAdminSettings</code></td>
<td>Define how an administrated service/daemon is remotely accessed via REST</td>
<td>2484</td>
</tr>
<tr>
<td><code>TDDDAdministratedDaemonSettings</code></td>
<td>Parent class for storing a service/daemon settings</td>
<td>2485</td>
</tr>
<tr>
<td><code>TDDDAppSettingsAbstract</code></td>
<td>Abstract class for storing application settings</td>
<td>2481</td>
</tr>
<tr>
<td><code>TDDDAppSettingsRest</code></td>
<td>Parent class for storing REST-based application settings</td>
<td>2484</td>
</tr>
<tr>
<td><code>TDDDAppSettingsStorageAbstract</code></td>
<td>Abstract parent class for storing application settings</td>
<td>2480</td>
</tr>
<tr>
<td><code>TDDDAppSettingsStorageFile</code></td>
<td>Class used for storing application settings as a JSON file</td>
<td>2482</td>
</tr>
<tr>
<td><code>TDDDLogSettings</code></td>
<td>Settings used to define how logging take place</td>
<td>2479</td>
</tr>
<tr>
<td>Objects</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TDDDMongoDBRestSettings</td>
<td>Storage class for a remote MongoDB server direct access settings</td>
<td>2488</td>
</tr>
<tr>
<td>TDDDHttpLogSettings</td>
<td>Stand-alone property to publish a secondary logged service over HTTP</td>
<td>2488</td>
</tr>
<tr>
<td>TDDDHttpSettings</td>
<td>Stand-alone property to publish a secondary TSQLRestServer over HTTP</td>
<td>2488</td>
</tr>
<tr>
<td>TDDDSettings</td>
<td>Storage class for initializing an ORM REST class</td>
<td>2482</td>
</tr>
<tr>
<td>TDDSLogRestSettings</td>
<td>Storage class for a ServicesLog settings</td>
<td>2487</td>
</tr>
<tr>
<td>TDDSocketThreadSettings</td>
<td>The settings of a TDDDThreadSocketProcess thread</td>
<td>2486</td>
</tr>
</tbody>
</table>

**TDDDLogSettings = class(TSynPersistent)**

*Settings used to define how logging take place*
- will map the most used TSynLogFamily parameters

**constructor** Create; **override**;

*Initialize the settings to their (TSynLogFamily) default values*

**property** AutoFlushTimeout: integer *read* fAutoFlush *write* fAutoFlush;

*The time (in seconds) after which the log content must be written on disk, whatever the current content size is*
- by default, the log file will be written for every 4 KB of log (see TSynLogFamily.BufferSize property) - this will ensure that the main application won't be slow down by logging
- in order not to loose any log, a background thread can be created and will be responsible of flushing all pending log content every period of time (e.g. every 10 seconds)

**property** ConsoleLevels: TSynLogInfos *read* fConsoleLevels *write* fConsoleLevels;

*The optional log levels to be used for the console*
- by default, only errors would be logged to the console
- you can specify here another set of levels, e.g. " star" for a verbose console output - note that console is very slow to write, so usually you should better not set a verbose definition here, unless you are in debugging mode

**property** CustomFileName: TFileName *read* fCustomFileName *write* fCustomFileName;

*Allows to customize the log file name*

**property** DestinationPath: TFileName *read* FDestinationPath *write* FDestinationPath;

*Allows to customize where the log files will be stored*

**property** Levels: TSynLogInfos *read* fLevels *write* fLevels;

*The log levels to be used for the log file*
- i.e. a combination of none or several logging event
- if " star" is serialized, unneeded sNone won't be part of the set
property LowLevelWebSocketsFrames: boolean read fLowLevelWebSocketsFrames write fLowLevelWebSocketsFrames;

If low-level WebSockets frames should be logged
- disabled by default, to minimize logged content
- may be enabled to monitor most (asynchronous) activity, especially in background threads

property RotateFileCount: cardinal read fRotateFileCount write fRotateFileCount;
Auto-rotation of logging files
- set to 0 by default, meaning no rotation

property RotateFileDailyAtHour: integer read fRotateFileDailyAtHour write fRotateFileDailyAtHour;
Fixed hour of the day where logging files rotation should be performed

property RotateFileSizeKB: cardinal read fRotateFileSize write fRotateFileSize;
Maximum size of auto-rotated logging files, in kilo-bytes (per 1024 bytes)

property StackTraceViaAPI: boolean read FStackTraceViaAPI write FStackTraceViaAPI;
By default (false), logging will use manual stack trace browsing
- if you experiment unexpected EAccessViolation, try to set this setting to TRUE so that the RtlCaptureStackBackTrace() API would be used instead

property SyslogFacility: TSyslogFacility read fSyslogFacility write fSyslogFacility;
The optional log levels to be used for remote UDP syslog server sending
- works in conjunction with SyslogServer/SyslogLevels properties
- default is sfLocal0

property SyslogLevels: TSynLogInfos read fSyslogLevels write fSyslogLevels;
The optional log levels to be used for remote UDP syslog server sending
- works in conjunction with SyslogServer property
- default will transmit all warnings, errors and exceptions

property SyslogServer: RawUTF8 read fSyslogServer write fSyslogServer;
The optional remote UDP syslog server
- expecting https://tools.ietf.org/html/rfc5424 messages over UDP
- e.g. ‘1.2.3.4’ to connect to UDP server 1.2.3.4 using default port 514 - but you can specify an alternative port as ‘1.2.3.4:2514’
- works in conjunction with SyslogLevels/SyslogFacility properties
- default is '' to disable syslog remote logging

TDDDAppSettingsStorageAbstract = class(TSynAutoCreateFields)
Abstract parent class for storing application settings

constructor Create(const aInitialJSON: RawUTF8); reintroduce; virtual;
Initialize the storage instance

procedure Store(const aJSON: RawUTF8); virtual;
TDDDAppSettingsAbstract would use this to actually persist the data
property InitialJsonContent: RawUTF8 read fInitialJsonContent;

The JSON content, as specified when creating the instance
- will allow SettingsDidChange to check if has changed
- here the JSON content is stored with default ObjectToJSON() options, so will be the normalized representation of the content, which may not match the JSON supplied to SetInitialJsonContent() protected method

property Owner: TDDDAppSettingsAbstract read fOwner;

The associated settings values

TDDDAppSettingsAbstract = class(TInterfacedObjectAutoCreateFields)

Abstract class for storing application settings
- this class implements IAutoCreateFieldsResolve so is able to inject its own values to any TInjectableAutoCreateFields instance
- you have to manage instance lifetime of these inherited classes with a local IAutoCreateFieldsResolve variable, just like any TInterfaceObject

constructor Create(aStorage: TDDDAppSettingsStorageAbstract); reintroduce;
Initialize the settings, with a corresponding storage process

destructor Destroy; override;
Persist if needed, and finalize the settings

function AsJson: RawUTF8; virtual;
Serialize the settings as JSON
- any enumerated or set published property will be commented with their textual values, and
'stored false' properties would be included
- returns the new JSON content corresponding to the updated settings

class function PasswordFields: RawUTF8;

Low-level method returning all TSynPersistentPassword full paths of all previously created TDDDAppSettingsStorageFile .settings
- as settingsfile=class1@full.path.to.pass1,class2@full.path.to.pass2,...
- you may use this method to create a 'passwords' resource for /HardenPasswords command line switch as implemented in dddInfraSettings.pas:
  passwords := SynLZCompress(TDDDAppSettingsAbstract.PasswordFields);
  FileFromString(passwords, 'passwords.data');

then create e.g. a passwords.rc file as such:
  passwords 10 "passwords.data"

  compile this resource:
  brc32 passwords.rc

  and link the resulting .res file to your daemon executable:
  {$R passwords.res}

then /HardenPasswords and /PlainPasswords command line switches will cypher/uncypher all TSynPersistentPassword protected fields using safe per-user CryptDataForCurrentUser() encryption

procedure Initialize(const aDescription: string); virtual;
To be called when the application starts, to initialize settings
- you can specify a default Description value
- it will set the global SQLite3Log.Family according to Log values
procedure StoreIfUpdated; virtual;
    Persist the settings if needed
    - just a wrapper around Storage.Store(AsJson)
    - implements IDDDSettingsStorable for "#settings save" admin command

property Description: string read FDescription write FDescription;
    Some text which will be used to describe this application

property Log: TDDDLogSettings read fLog;
    Defines how logging will be done for this application

property Storage: TDDDAppSettingsStorageAbstract read fStorage;
    Access to the associated settings storage

property SyslogProcID: RawUTF8 read fSyslogProcID write fSyslogProcID;
    Transmitted as PROCID as part of any Log.SyslogServer message

TDDDAppSettingsStorageFile = class(TDDDAppSettingsStorageAbstract)
    Class used for storing application settings as a JSON file

constructor Create(const aSettingsJsonFileName: TFileName=''); reintroduce;
    virtual;
    Initialize and read the settings from the supplied JSON file name
    - if no file name is specified, will use the executable name with '.settings' as extension

function FileNameRelativeToSettingsFile(const aFileName: TFileName): TFileName;
    Compute a file name relative to the .settings file path

property SettingsJsonFileName: TFileName read fSettingsJsonFileName write fSettingsJsonFileName;
    The .settings file name, including full path

TDDLDRestSettings = class(TSynAutoCreateFields)
    Storage class for initializing an ORM REST class
    - this class will contain some generic properties to initialize a TSQLRest pointing to a local or remote SQL/NoSQL database, with optional wrappers

function NewRestInstance(aRootSettings: TDDDAppSettingsAbstract; const aTables: array of TSQLRecordClass; aOptions: TDDDNewRestInstanceOptions; aExternalDBOptions: TVirtualTableExternalRegisterOptions=[regDoNotRegisterUserGroupTables]; aMongoDBIdentifier: word=0; aMongoDBOptions: TStaticMongoDBRegisterOptions=[mrDoNotRegisterUserGroupTables]): TSQLRest;
    overload; virtual;
    Is able to instantiate a REST instance according to the stored definition
    - just an overloaded version which will create an owned TSQLModel with the supplied TSQLRecord classes
function NewRestInstance(aRootSettings: TDDDApSettingsAbstract; aModel: TSQLModel; aOptions: TDDDDNewRestInstanceOptions; aExternalDBOptions: TVirtualTableExternalRegisterOptions=[regDoNotRegisterUserGroupTables]; aMongoDBIdentifier: word=0; aMongoDBOptions: TStaticMongoDBRegisterOptions=[mrDoNotRegisterUserGroupTables]): TSQLRest; overload; virtual;

Is able to instantiate a REST instance according to the stored definition
- Definition.Kind will identify the TSQLRestServer or TSQLRestClient class to be instantiated, or if equals 'MongoDB'/'MongoDBS' use a full MongoDB engine, or an external SQL database if it matches a TSQLDBConnectionProperties classname
- if aDefaultLocalSQLi3e3 is TRUE, then if Definition.Kind is '', a local SQLite file database will be initiated
- if aMongoDBIdentifier is not 0, then it will be supplied to every TSQLRestStorageMongoDB.SetEngineAddCompuIdentifier() created
- will return nil if the supplied Definition is not correct
- note that the supplied Model.Root is expected to be the default root URI, which will be overridden with this TDDDRestSettings.Root property
- will also publish /wrapper HTML page if WrapperTemplateFolder is set

function NewRestServerDB(const aDBFileName: TFileName; const aModelRoot: RawUTF8; const aModelTables: array of TSQLRecordClass; aOptions: TDDDRestSettingsOptions=[]; aCacheSize: cardinal=10000): TSQLRestServerDB;

Initialize a stand-alone TSQLRestServerDB instance
- with its own database file located in DefaultDataFileName + aDBFileName
- will own its own TSQLModel with aModelRoot/aModelTables
- you can tune aCacheSize if the default 40MB value is not right
- will eventually call CreateMissingTables
- define custom TDDDRestSettingsOptions if needed

function WrapperSourceFolderFixed: TFileName;

Returns the WrapperSourceFolder property, all / chars replaced by \\ so that you would be able to store the paths with /, avoiding JSON escape

function WrapperTemplateFolderFixed(ReturnLocalIfNoneSet: boolean=false): TFileName;

Returns the WrapperTemplateFolder property, all / chars replaced by \\ so that you would be able to store the paths with /, avoiding JSON escape

class procedure RestServerDBSetOptions(DB: TSQLRestServer; Options: TDDDRestSettingsOptions);

If DB is a TSQLRestServerDB, will define the expection options
- DB.FileName will be erased from disk if optEraseDBFileAtStartup is defined
- force LockingMode=exclusive and Synchronous=off unless optSQLite3FileSafeNonExclusive/optSQLite3FileSafeSlowMode options are set

procedure WrapperGenerate(Rest: TSQLRestServer; Port: integer; const DestFile: TFileName; const Template: TFileName = 'API.adoc.mustache');

Generate API documentation corresponding to REST SOA interfaces

property DefaultDataFileName: RawUTF8 read fDefaultDataFileName write fDefaultDataFileName;
The default database file name
**property** DefaultDataFolder: TFileName read fDefaultDataFolder write fDefaultDataFolder;

*The default folder where database files are to be stored*
- will be used by NewRestInstance instead of the .exe folder, if set

**property** Options: TDDDRestSettingsOptions read fOptions write fOptions;

*How the REST instance is to be initialized*

**property** ORM: TSynConnectionDefinition read fORM;

*Defines a mean of access to a TSQLRest instance*
- using Kind/ServerName/DatabaseName/User properties: Kind would define the TSQLRest class to be instantiated by function NewRestInstance()

**property** Root: RawUTF8 read fRoot write fRoot;

*The URI Root to be used for the REST Model*

**property** WrapperSourceFolders: TFileName read fWrapperSourceFolders write fWrapperSourceFolders;

*Where the source code may be searched, for comment extraction of types*
- several folders may be defined, separated by ; (just like in Delphi IDE)
- only used if WrapperTemplateFolder is defined

**property** WrapperTemplateFolder: TFileName read fWrapperTemplateFolder write fWrapperTemplateFolder;

*If set to a valid folder, the generated TSQLRest will publish a '/Root/wrapper' HTML page so that client code could be generated*

```plaintext
TDDDApptSettingsRest = class(TDDDApptSettingsAbstract)
Parent class for storing REST-based application settings
- this class could be used for an application with a single REST server running on a given HTTP port

procedure Initialize(const aDescription: string); override;
To be called when the application starts, to initialize settings
- will call inherited TDDDApptSettingsFile.Initialize, and set ServerPort to a default 888/8888 value under Windows/Linux

**property** Rest: TDDDRestSettings read fRest;
Allow to instantiate a REST instance from its JSON definition

**property** ServerPort: RawUTF8 read fServerPort write fServerPort;
The IP port to be used for the HTTP server associated with the application
```

```plaintext
TDDDAdministratedDaemonRemoteAdminSettings = class(TSynAutoCreateFields)
Define how an administrated service/daemon is remotely accessed via REST
- the IAdministratedDaemon service will be published to administrate this service/daemon instance
- those values should match the ones used on administrative tool side
```
property AuthHashedPassword: RawUTF8 read FAuthHashedPassword write FAuthHashedPassword stored false;
The SHA-256 hashed password to authenticate AuthUserName
- follows the TSQLAuthUser.ComputeHashedPassword() encryption
- marked as 'stored false' so that it won't appear e.g. in the logs

property AuthHttp: TSQLHttpServerDefinition read FAuthHttp;
If defined, these parameters would be used for REST publishing over HTTP

property AuthNamedPipeName: TFileName read FAuthNamedPipeName write FAuthNamedPipeName;
If defined, the following pipe name would be used for REST publishing
- by definition, will work only on Windows

property AuthRootURI: RawUTF8 read FAuthRootURI write FAuthRootURI;
The root URI used for the REST data model
- default URI is 'admin'

property AuthUserName: RawUTF8 read FAuthUserName write FAuthUserName;
If set, expect authentication with this single user name
- that is, the TSQLRestServer will register a single TSQLAuthUser instance with the supplied
AuthUserName/AuthHashedPassword credentials

TDDDAdministratedDaemonSettings = class(TDDDAppSettingsAbstract)
Parent class for storing a service/daemon settings
- under Windows, some Service* properties will handle installation as a regular Windows Service,
thanks to TDDDDaemon

function SettingsFolder: TFileName; virtual;
Returns the folder containing .settings files -.exe folder by default

procedure Initialize(const aDescription, aServiceName,aServiceDisplayName,aAppUserModelID: string; const aServiceDependencies: TStringDynArray = nil); reintroduce; virtual;
To be called when the application starts, to initialize settings
- you can specify default Description and Service identifiers
- the service-related parameters are Windows specific, and will be ignored on other platforms

property AppUserModelID: string read FAppUserModelID write FAppUserModelID;
Under Windows 7 and later, will set an unique application-defined Application User Model ID
(AppUserModelID) that identifies the current process to the taskbar
- this identifier allows an application to group its associated processes and windows under a
single taskbar button
- should follow SetAppUserModelID() expectations, i.e. 'Company.Product'

property RemoteAdmin: TDDDAdministratedDaemonRemoteAdminSettings read FRemoteAdmin;
Define how this administrated service/daemon is accessed via REST

property ServiceAutoStart: boolean read FServiceAutoStart write FServiceAutoStart;
Under Windows, will define if the Service should auto-start at boot
- FALSE means that it should be started on demand
property ServiceDependencies: TStringDynArray read FServicDependencies write FServicDependencies;

Under Windows, will define optional Service internal Dependencies
- not published by default: could be defined if needed, or e.g. set in overriden constructor

property ServiceDisplayName: string read FServicDisplayName write FServicDisplayName;

Under Windows, will define the Service displayed name

property ServiceName: string read FServicename write FServicename;

Under Windows, will define the Service internal name

TDDDSocketThreadSettings = class(TSynAutoCreateFields)
The settings of a TDDDSocketThreadProcess thread
- defines how to connect (and reconnect) to the associated TCP server

constructor Create; override;
Used to set the default values

function GetHostPort: RawUTF8;
Retrieve Host and Port values as a single 'ip:port' text

function SetHostPort(const IpPort: RawByteString; defaultPort: integer): boolean;
Set Host and Port values from a 'ip:port' or 'ip' text

property AutoReconnectAfterSocketError: boolean read FAutoReconnectAfterSocketError write FAutoReconnectAfterSocketError;
If TRUE, any communication error would try to reconnect the socket

property ConnectionAttemptsInterval: Integer read fConnectionAttemptsInterval write fConnectionAttemptsInterval;
The time, in seconds, between any reconnection attempt
- default value is 5 - i.e. five seconds
- if you set -1 as value, thread would end without any retrial

property Host: RawUTF8 read FHost write FHost;
The associated TCP server host

property MonitoringLogInterval: integer read FMonitoringInterval write FMonitoringInterval;
The period, in milliseconds, on which Monitoring information is logged
- default value is 120000, i.e. 2 minutes

property OnIDDSSocketThreadCreate: TOnIDDSSocketThreadCreate read fOnIDDSSocketThreadCreate write fOnIDDSSocketThreadCreate;
You could set here a factory method to mock the socket connection
- this property is public, but not published, since it should not be serialized on the settings file, but overloaded at runtime

property Port: integer read FPort write FPort;
The associated TCP server port
**property** SocketBufferBytes: integer *read* FSocketBufferBytes *write* FSocketBufferBytes;

*The internal size of the input socket buffer*
- default is 32768, i.e. 32 KB

**property** SocketLoopPeriod: integer *read* FSocketLoopPeriod *write* FSocketLoopPeriod;

*How many millisecond the main socket reading loop should wait for pending data, before calling TDDDSocketThread.InternalExecuteIdle*
- default is 100 ms

**property** SocketMaxBufferBytes: integer *read* FSocketMaxBufferBytes *write* FSocketMaxBufferBytes;

*The maximum size of the thread input buffer*
- i.e. how many bytes are stored in fSocketInputBuffer memory, before nothing is retrieved from the socket buffer
- set to avoid any "out of memory" of the current process, if the incoming data is not processed fast enough
- default is 16777216, i.e. 16 MB

**property** SocketTimeout: integer *read* FSocketTimeout *write* FSocketTimeout;

*The time out period, in milliseconds, for socket access*
- default is 2000 ms, i.e. 2 seconds

TDDDDServicesLogRestSettings = class(TDDDRestSettings)

*Storage class for a ServicesLog settings*

**function** NewRestInstance(aRootSettings: TDDDAppSettingsAbstract; aMainRestWithServices: TSQLRestServer; const aLogClass: array of TSQLRecordServiceLogClass; const aExcludedMethodNamesCSV: RawUTF8; aShardRange: TID=50000): TSQLRest; reintroduce;

*Compute a stand-alone REST instance for interface-based services logging*
- all services of aMainRestWithServices would log their calling information into a dedicated table, but the methods defined in aExcludedMethodNamesCSV (which should be specified, even as ",", to avoid FPC compilation error)
- by default, will create a local SQLite3 file for storage, optionally via TSQLRestStorageShardDB if ShardDBCount is set
- the first supplied item of aLogClass array would be used for the service logging; any additional item would be part of the model of the returned REST instance, but may be used later on (e.g. to handle DB-based asynchronous remote notifications as processed by TServiceFactoryClient.SendNotificationsVia method)
- if aLogClass=[], plain TSQLRecordServiceLog would be used as default
- aShardRange is used for TSQLRestStorageShardDB if ShardDBCount>0

**property** ShardDBCount: Integer *read* fShardDBCount *write* fShardDBCount;

*If set, will define MaxShardCount for TSQLRestStorageShardDB persistence*

TDDDAdministratedDaemonHttpSettings = class(TDDDAdministratedDaemonSettings)

*Parent class for storing a HTTP published service/daemon settings*
property Http: TSQLHttpServerDefinition read fHttp;
  How the HTTP server should be defined

property Rest: TDDDRestSettings read fRest;
  How the main REST server is implemented
  - most probably using a TSQLRestServerDB, i.e. local SQLite3 storage

property ServicesLog: TDDDServicesLogRestSettings read fServicesLog;
  How the SOA calls would be logged into their own SQLite3 database

TDDDRestHttpSettings = class(TSynAutoCreateFields)
  Stand-alone property to publish a secondary TSQLRestServer over HTTP

property Http: TSQLHttpServerDefinition read fHttp;
  How the HTTP server should be defined

property Rest: TDDDRestSettings read fRest;
  How the REST server is implemented
  - most probably using a TSQLRestServerDB, i.e. local SQLite3 storage

TDDDRestLogSettings = class(TDDDRestHttpSettings)
  Stand-alone property to publish a secondary logged service over HTTP

property ServicesLog: TDDDServicesLogRestSettings read fServicesLog;
  How the SOA calls would be logged into their own SQLite3 database

TDDDMongoDBRestSettings = class(TDDDRestSettings)
  Storage class for a remote MongoDB server direct access settings

procedure SetDefaults(const Root, MongoServerAddress, MongoDatabase, MongoUser, MongoPassword: RawUTF8; TLS: boolean=false);
  Set the default values for direct MongoDB server connection
  - if MongoServerAddress is e.g. '?:27017', entry with default value would be saved in the settings, but NewRestInstance() would ignore it: once the remote MongoDB server IP is known, you may just replace '?' to use it
  - if MongoUser and MongoPassword are set, would call TMongoClient.OpenAuth()

Types implemented in the dddInfraSettings unit

TDDDAppSettingsAbstractClass = class of TDDDAppSettingsAbstract;
  Class type used for storing application settings

TDDDNewRestInstanceOptions = set of ( riOwnModel, riCreateVoidModelIfNone, riHandleAuthentication, riDefaultLocalSQLite3IfNone, riDefaultInMemorySQLite3IfNone, riDefaultFullMemoryIfNone, riDefaultLocalBinaryFullMemoryIfNone, riCreateMissingTables, riRaiseExceptionIfNoRest, riWithInternalState);
  How TDDDRestSettings.NewRestInstance would create its instances
  - riOwnModel will set ModellInstance.Owner := RestInstance
  - riHandleAuthentication will set the corresponding parameter to true
  - riDefaultLocalSQLite3IfNone/riDefaultInMemorySQLite3IfNone will create a SQLite3 engine with a
local file/memory storage, if TDDDRestSettings.ORM.Kind is not set
- riDefaultFullMemoryIfNone will create a TSQLRestServerFullMemory non persistent storage, or
riDefaultLocalBinaryFullMemoryIfNone with a binary local file, if TDDDRestSettings.ORM.Kind is not set
- riCreateMissingTables will call RestInstance.CreateMissingTables
- riRaiseExceptionIfNoRest will raise an EDDDInfraException if TDDDRestSettings.NewRestInstance
would return nil
- riWithInternalState will enable 'Server-InternalState:' header transmission i.e. disable
rsoNoInternalState for TSQLRestServer.Options

TDDDRestSettingsOption = ( optEraseDBFileAtStartup, optSQLite3FileSafeSlowMode,
optSQLite3FileSafeNonExclusive, optNoSystemUse, optSQLite3File4MBCacheSize,
optForceAjaxJson, optSQLite3LogQueryPlan );

Some options to be used for TDDDRestSettings
- as part of the .settings, they may be tuned for specific installations, whereas
TDDDNewRestInstanceOptions are defined in code

TDDDRestSettingsOptions = set of TDDDRestSettingsOption;

Define options to be used for TDDDRestSettings

TOnIDDSSocketThreadCreate = procedure(aOwner: TObject; out Obj: object);

A Factory event allowing to customize/mock a socket connection
- the supplied aOwner should be a TDDDSocketThread instance
- returns a IDDDSsocket interface instance (e.g. a TDDDSynCrtSocket)
27.86. SynDBBDE.pas unit

Purpose: BDE access classes for SynDB units
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the SynDBBDE unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynDBDataset</td>
<td>DB.pas TDataset-based direct access classes (abstract TQuery-like)</td>
<td>1267</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

```
SynDBBDE class hierarchy

ESQLDBBDE Dataset --> ESQLDBBDE

TSQLDBConnectionThreadSafe --> TSQLDBBDEConnection

TSQLDBDatasetConnectionProperties --> TSQLDBBDEConnectionProperties

TSQLDBDatasetStatement --> TSQLDBBDEStatement
```

Objects implemented in the SynDBBDE unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESQLDBBDE</td>
<td>Exception type associated to the direct BDE connection</td>
<td>2491</td>
</tr>
<tr>
<td>TSQLDBBDEConnection</td>
<td>Implements a direct connection via the BDE access layer</td>
<td>2491</td>
</tr>
<tr>
<td>TSQLDBBDEConnectionProperties</td>
<td>Implement properties shared by BDE connections</td>
<td>2491</td>
</tr>
<tr>
<td>TSQLDBBDEStatement</td>
<td>Implements a statement via a BDE connection</td>
<td>2492</td>
</tr>
</tbody>
</table>
ESQLDBBDE = class(ESQLDBDataset)

Exception type associated to the direct BDE connection

TSQDLBBDEConnectionProperties = class(TSQLDBDatasetConnectionProperties)

Implement properties shared by BDE connections

constructor Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8); override;

Initialize the properties to connect to the BDE engine
- aServerName shall contain the BDE Alias name
- aDatabaseName is ignored

function NewConnection: TSQLDBConnection; override;

Create a new connection
- caller is responsible of freeing this instance
- this overridden method will create an TSQLDBBDEConnection instance

TSQDLBBDEConnection = class(TSQLDBConnectionThreadSafe)

Implements a direct connection via the BDE access layer

constructor Create(aProperties: TSQLDBConnectionProperties); override;

Prepare a connection to a specified BDE database server

destructor Destroy; override;

Release memory and connection

function IsConnected: boolean; override;

Return TRUE if Connect has been already successfully called

function NewStatement: TSQLDBStatement; override;

Create a new statement instance

procedure Commit; override;

Commit changes of a Transaction for this connection
- StartTransaction method must have been called before

procedure Connect; override;

Connect to the specified BDE server
- should raise an ESQLDBBDE on error

procedure Disconnect; override;

Stop connection to the specified BDE database server
- should raise an ESQLDBBDE on error

procedure Rollback; override;

Discard changes of a Transaction for this connection
- StartTransaction method must have been called before

procedure StartTransaction; override;

Begin a Transaction for this connection
property Database: TDatabase read fDatabase;
Access to the associated BDE connection instance

property DBMS: TSQLDBDefinition read fDBMS;
The remote DBMS type, as retrieved at BDE connection creation

property DBMSName: RawUTF8 read fDBMSName;
The remote DBMS name, as retrieved at BDE connection creation

TSQLDBBDEStatement = class(TSQLDBDatasetStatement)
Implements a statement via a BDE connection
27.87. SynDBFireDAC.pas unit

*Purpose*: FireDAC/AnyDAC-based classes for SynDB units
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynDBFireDAC unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td>SynDBDataset</td>
<td>DB.pas TDataset-based direct access classes</td>
<td>1267</td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features as a complement to SynCommons, which tended to increase too much licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1721</td>
</tr>
</tbody>
</table>

**SynDBFireDAC class hierarchy**

**Objects implemented in the SynDBFireDAC unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESQLDBFireDAC</td>
<td>Exception type associated to FireDAC/AnyDAC database access</td>
<td>2494</td>
</tr>
<tr>
<td>TSQLDBFireDACConnecti on</td>
<td>implements a direct connection via FireDAC/AnyDAC database access</td>
<td>2495</td>
</tr>
<tr>
<td>TSQLDBFireDACConnecti onProperties</td>
<td>connection properties definition using FireDAC/AnyDAC database access</td>
<td>2494</td>
</tr>
</tbody>
</table>
ESQLDBFireDAC = class(ESQLDBDataset)

Exception type associated to FireDAC/AnyDAC database access

TSQLDBFireDACConnectionProperties = class(TSQLDBDatasetConnectionProperties)

connection properties definition using FireDAC/AnyDAC database access

constructor Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8);

Initialize the properties to connect via FireDAC/AnyDAC database access
- aServerName shall contain the FireDAC provider DriverID, e.g. 'Ora', and some optional parameters (e.g. remote server name if needed), after a '?' and separated by ';'
  - for instance:
    Create('Ora', 'TNSNAME', 'User', 'Password');
    Create('Ora?CharacterSet=cl8mswin1251', 'TNSNAME', 'User', 'Password');
    Create('MYSQL?Server=127.0.0.1\SQLEXPRESS', 'Northwind', 'User', 'Password');
    Create('MYSQL?Server=.\SQLEXPRESS;OSAuthent=Yes', '', '', '');
    Create('MSAcc', 'c:\data\access.mdb', '', '');
    Create('MySQL?Server=127.0.0.1;Port=3306', 'MyDB', 'User', 'Password');
    Create('SQLite', 'c:\data\myapp.db3', '', '');
    Create('SQLite', 'c:\data\myapp.db3', '','');
    Create('IB', '127.0.0.1:C:\IB\IBDEMO_IB2007.IB', 'User', 'Password');
    Create('IB?Server=my_host/3055', 'C:\ib\ADDEMO_IB2007.IB', 'User', 'Password');
    Create('IB?CreateDatabase=Yes', '127.0.0.1:C:\ib\ADDEMO_IB2007.IB', 'User', 'Password');
    Create('DB2?Server=localhost/3055', 'SAMPLE', 'db2admin', 'db2Password');
    Create('PG?Server=localhost;Port=5432', 'postgres', 'postgres', 'postgresPassword');
    Create('MySQL?Server=localhost;Port=3306', 'test', 'root', '');
- aDatabaseName shall contain the database server name
- note that you need to link the FireDAC driver by including the expected uADPhys*.pas / FireDAC.Phys.*.pas units into a uses clause of your application, e.g. uADPhysOracle, uADPhysMSSQl, uADPhysMSAcc, uADPhysMySQL, uADPhysSQLite, uADPhysIB or uADPhysDB2 (depending on the expected provider) - or FireDAC.Phys.Oracle, FireDAC.Phys.MSAcc, FireDAC.Phys.MSSQl, FireDAC.Phys.SQLite, FireDAC.Phys.IB, FireDAC.Phys.PG or FireDAC.Phys.DBD2 since Delphi XE5 namespace modifications

destructor Destroy; override;

Release internal structures

function NewConnection: TSQLDBConnection; override;

Create a new connection
- caller is responsible of freeing this instance
- this overridden method will create an TSQLDBFireDACConnection instance

procedure GetFields(const aTableName: RawUTF8; out Fields: TSQLDBColumnDefineDynArray); override;

Retrieve the column/field layout of a specified table
- this overridden method will use FireDAC metadata to retrieve the information
procedure GetIndexes(const aTableName: RawUTF8; out Indexes: TSQLDBIndexDefineDynArray); override;

Retrieve the advanced indexed information of a specified Table
- this overridden method will use FireDAC metadata to retrieve the information

procedure GetTableNames(out Tables: TRawUTF8DynArray); override;

Get all table names
- this overridden method will use FireDAC metadata to retrieve the information

property Parameters: TStringList read fFireDACOptions;

Allow to set the options specific to a FireDAC driver
- by default, ServerName, DatabaseName, UserID and Password are set by the Create() constructor according to the underlying FireDAC driver
- you can add some additional options here

TSQLDBFireDACConnection = class(TSQLDBConnectionThreadSafe)

\text{implements a direct connection via FireDAC/AnyDAC database access}

constructor Create(aProperties: TSQLDBConnectionProperties); override;

\text{Prepare a connection for a specified FireDAC/AnyDAC database access}

destructor Destroy; override;

\text{Release memory and connection}

function IsConnected: boolean; override;

\text{Return TRUE if Connect has been already successfully called}

function NewStatement: TSQLDBStatement; override;

\text{Create a new statement instance}

procedure Commit; override;

\text{Commit changes of a Transaction for this connection}
- StartTransaction method must have been called before

procedure Connect; override;

\text{Connect to the specified database server using FireDAC}
- should raise an ESQLDBFireDAC on error

procedure Disconnect; override;

\text{Stop connection to the specified database server using FireDAC}
- should raise an ESQLDBFireDAC on error

procedure Rollback; override;

\text{Discard changes of a Transaction for this connection}
- StartTransaction method must have been called before

procedure StartTransaction; override;

\text{Begin a Transaction for this connection}

property Database: TADConnection read fDatabase;

\text{Access to the associated FireDAC connection instance}
TSQLDBFireDACStatement = class(TSQLDBDatasetStatementAbstract)
  implements a statement via a FireDAC connection
- this specific version will handle the FireDAC specific parameter classes
- it will also handle Array DML commands, if possible

procedure Prepare(const aSQL: RawUTF8; ExpectResults: boolean = false); overload;
override;
  Prepare an UTF-8 encoded SQL statement
- parameters marked as ? will be bound later, before ExecutePrepared call
- if ExpectResults is TRUE, then Step() and Column*() methods are available to retrieve the data rows
- raise an ESQLDBFireDAC on any error

Constants implemented in the SynDBFireDAC unit

FIREDAC_PROVIDER: array[doOracle..high(TSQLDBDefinition)] of RawUTF8 = ('Ora','MSSQL','MSAcc','MySQL','SQLite','IB','','PG','DB2','Infx');

FireDAC DriverID values corresponding to SynDB recognized SQL engines
27.88. SynDBNexusDB.pas unit

**Purpose**: NexusDB 3.x direct access classes (embedded engine only)
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynDBNexusDB unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>SynCommons</em></td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>version 1.18</td>
<td></td>
</tr>
<tr>
<td><em>SynDB</em></td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>version 1.18</td>
<td></td>
</tr>
<tr>
<td><em>SynDBDataset</em></td>
<td>DB.pas TDataset-based direct access classes (abstract TQuery-like)</td>
<td>1267</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>version 1.18</td>
<td></td>
</tr>
<tr>
<td><em>SynLog</em></td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>version 1.18</td>
<td></td>
</tr>
<tr>
<td><em>SynTable</em></td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td></td>
<td>- as a complement to SynCommons, which tended to increase too much</td>
<td></td>
</tr>
<tr>
<td></td>
<td>licensed under a MPL/GPL/LGPL tri-license;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

**SynDBNexusDB class hierarchy**

**Objects implemented in the SynDBNexusDB unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ESQLDBNexusDB</em></td>
<td>Exception type associated to the direct NexusDB connection</td>
<td>2498</td>
</tr>
<tr>
<td><em>TSQLDBNexusDBConnection</em></td>
<td>Implements a direct connection to the native NexusDB database</td>
<td>2499</td>
</tr>
<tr>
<td><em>TSQLDBNexusDBConnectionProperties</em></td>
<td>Implement properties shared by native NexusDB connections</td>
<td>2498</td>
</tr>
</tbody>
</table>
ESQLDBNexusDB = class(ESQLDBDataset)

(Exception type associated to the direct NexusDB connection)

TSQLDBNexusDBConnectionProperties =
class(TSQLDBDatasetConnectionProperties)

(Implement properties shared by native NexusDB connections)
- note that only the embedded engine is implemented by now - feedback needed!

constructor Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8);

Initialize the properties to connect to the NexusDB engine
- this overridden method will initialize the protocol to be used as stated by aServerName i.e. nxpTCP://11.23.34.43
- Default protocol is nxpFolder
- if protocol is nxpFolder then aDatabaseName will contain the path to the folder to be used
- if protocol is other then nxpFolder than aServerName will contain the server to connect to and aDatabaseName will contains the alias of the database
- Possible aServerName formats:
  <protocol>://<servername>/<alias>  (aDatabaseName wil be overwritten by this alias)
  <protocol>://servername            (aDatabaseName will contain alias)
  ''                                 (aDatabaseName contains path to nxpFOLDER database)

function ColumnTypeNativeToDB(const aNativeType: RawUTF8; aScale: integer):
TSQLDBFieldType; override;

Convert a textual column data type, as retrieved e.g. from SQLGetField, into our internal primitive types

function CreateDatabase: boolean; virtual;

Create the database folder (if not existing)

function DatabaseExists: boolean; virtual;

Determine if database exists
- just test if the corresponding folder exists

function DeleteDatabase: boolean; virtual;

Delete the database folder
- including all its files - so to be used carefully!

function NewConnection: TSQLDBConnection; override;

Create a new connection
- caller is responsible of freeing this instance
- this overridden method will create an TSQLDBNexusDBConnection instance

property Protocol: TNXProtocol read fProtocol;

The transport protocol used to connect to the NexusDB engine
TSQLDBNexusDBConnection = class(TSQLDBConnectionThreadSafe)

  Implements a direct connection to the native NexusDB database

constructor Create(aProperties: TSQLDBConnectionProperties); override;
  Prepare a connection to a specified NexusDB database server

destructor Destroy; override;
  Release memory and connection

function IsConnected: boolean; override;
  Return TRUE if Connect has been already successfully called

function NewStatement: TSQLDBStatement; override;
  Create a new statement instance

procedure Commit; override;
  Commit changes of a Transaction for this connection
  - StartTransaction method must have been called before

procedure Connect; override;
  Connect to the specified NexusDB server
  - should raise an ESQLDBNexusDB on error

procedure Disconnect; override;
  Stop connection to the specified NexusDB database server
  - should raise an ESQLDBNexusDB on error

procedure Rollback; override;
  Discard changes of a Transaction for this connection
  - StartTransaction method must have been called before

procedure StartTransaction; override;
  Begin a Transaction for this connection

property Database: TnxDatabase read fDatabase;
  Access to the associated NexusDB connection instance

property ServerEngine: TnxBaseServerEngine read fServerEngine write SetServerEngine;
  Associated NexusDB server engine

TSQLDBNexusDBStatement = class(TSQLDBDatasetStatement)

  Implements a statement via the native NexusDB connection

Types implemented in the SynDBNexusDB unit

TNXProtocol = ( nxpUnknown, nxpFOLDER, nxpTCPIP, nxpPIPE, nxpCOM, nxpMEM, nxpBFISH );
  Available communication protocols used by NexusDB between client and server
  - nxpFOLDER: default protocol, accessing NexusDB database in a Windows Folder
  - nxpTCPIP: TCP/IP transport, indicated by nxtcp://
  - nxpPIPE: Windows Named Pipe transport, indicated by nxpipe://
  - nxpMEM: direct memory transport, indicated by nxmem://
  - nxpBFISH: BlowFish transport, indicated by nxbfisch://
### Constants implemented in the `SynDBNexusDB` unit

```plaintext
NEXUSDB_INMEMORY = '#INMEM';
```

*Set `aServerName` to this value to create an in-memory table*
- do not use this constant, since it was not working as expected yet

### Functions or procedures implemented in the `SynDBNexusDB` unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DropNexusEmbeddedEngine</code></td>
<td>Release any internal NexusDB embedded engine</td>
<td>2500</td>
</tr>
<tr>
<td><code>GetNXProtocol</code></td>
<td>Determine NexusDB transport protocol (TNXProtocol) to be used, based on protocol indicator in connection string</td>
<td>2500</td>
</tr>
<tr>
<td><code>NexusEmbeddedEngine</code></td>
<td>Return the internal NexusDB embedded engine</td>
<td>2500</td>
</tr>
</tbody>
</table>

**Function** `DropNexusEmbeddedEngine`: `TnxServerEngine`;

*Release any internal NexusDB embedded engine*
- returns nil on success, or `PtrInt(-1)` if was not initialized

**Function** `GetNXProtocol`(`const` `aConnectionString`: `RawUTF8`; `out` `aServerName`: `RawUTF8`; `out` `aAlias`: `RawUTF8`): `TNXProtocol`;

*Determine NexusDB transport protocol (TNXProtocol) to be used, based on protocol indicator in connection string*
- if no protocol specifier is included in the connectionstring then `nxpFOLDER` is assumed.
- `aServerName` will contain the URL to the Server if the protocol is not `nxpFOLDER`

**Function** `NexusEmbeddedEngine`: `TnxServerEngine`;

*Return the internal NexusDB embedded engine*
- initialize it, if was not already the case
27.89. SynDBUniDAC.pas unit

**Purpose**: UniDAC-based classes for SynDB units
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

**Units used in the SynDBUniDAC unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynDB</td>
<td>Abstract database direct access classes</td>
<td>1204</td>
</tr>
<tr>
<td>SynDBDataset</td>
<td>DB.pas TDataset-based direct access classes (abstract TQuery-like)</td>
<td>1267</td>
</tr>
<tr>
<td>SynLog</td>
<td>Logging functions used by Synopse projects</td>
<td>1363</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
</tbody>
</table>

Objects implemented in the SynDBUniDAC unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESQldbUniDAC</td>
<td>Exception type associated to UniDAC database access</td>
<td>2502</td>
</tr>
<tr>
<td>TSQLDBUniDACConnection</td>
<td>implements a direct connection via UniDAC database access</td>
<td>2503</td>
</tr>
<tr>
<td>TSQLDBUniDACConnectionProperties</td>
<td>connection properties definition using UniDAC database access</td>
<td>2502</td>
</tr>
</tbody>
</table>
**Sympose mORMot Framework**  
**Software Architecture Design 1.18**  
**Date: September 16, 2020**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLDBUniDACStatement</td>
<td>implements a statement via a UniDAC connection</td>
<td>2504</td>
</tr>
</tbody>
</table>

**ESQLDBUniDAC** = **class**(ESQLDBDataset)

`Exception type associated to UniDAC database access`

**TSQLDBUniDACConnectionProperties** = **class**(TSQLDBDatasetConnectionProperties)

`connection properties definition using UniDAC database access`

**constructor** Create(const aServerName, aDatabaseName, aUserID, aPassWord: RawUTF8); override;

`Initialize the properties to connect via UniDAC database access`
- aServerName shall contain the UniDAC provider name, e.g. 'Oracle' - you can use the TSQLDBUniDACConnectionProperties.URI() to retrieve the provider name from its SynDB.TSQLDBDefinition enumeration, and optionally set some options, which will be added to the internal SpecificOptions[]:
  - 'Oracle?ClientLibrary=oci64\oci1.dll'
  - 'MySQL?Server=192.168.2.60;Port=3306', 'world', 'root', 'dev'
- aDatabaseName shall contain the database server name

**destructor** Destroy; override;

`Release internal structures`

**function** NewConnection: TSQLDBConnection; override;

`Create a new connection`
- caller is responsible of freeing this instance
- this overridden method will create an TSQLDBUniDACConnection instance

**class function** URI(aServer: TSQLDBDefinition; const aServerName: RawUTF8; const aLibraryLocation: TFileName=''; aLibraryLocationAppendExePath: boolean=true): RawUTF8;

`Compute the UniDAC URI from a given database engine and server name`
- the optional server name can contain a port number, specified after ':'
- you can set an optional full path to the client library name, to be completed on the left side with the executable path
- possible use may be:
  - PropsOracle := TSQLDBUniDACConnectionProperties.Create(
    TSQLDBUniDACConnectionProperties.URI(dOracle,'','oci64\oci1.dll'),
    'tnsname','user','pass');
  - PropsFirebird := TSQLDBUniDACConnectionProperties.Create(
    TSQLDBUniDACConnectionProperties.URI(dFirebird,'','Firebird\fbembed.dll'),
    'databasefilename','','');
  - PropsMySQL := TSQLDBUniDACConnectionProperties.Create(
    TSQLDBUniDACConnectionProperties.URI(dMySQL,'192.168.2.60:3306'),
    'world','root','dev');

**procedure** GetFields(const aTableName: RawUTF8; out Fields: TSQLDBColumnDefineDynArray); override;

`Retrieve the column/field layout of a specified table`
- this overridden method will use UniDAC metadata to retrieve the information
procedure GetIndexes(const aTableName: RawUTF8; out Indexes: TSQLDBIndexDefineDynArray); override;

*Retrieve the advanced indexed information of a specified Table*
- this overridden method will use UniDAC metadata to retrieve the information

procedure GetTableNames(out Tables: TRawUTF8DynArray); override;

*Get all table names*
- this overridden method will use UniDAC metadata to retrieve the information

property SpecificOptions: TStringList read fSpecificOptions;

*Allow to set the options specific to a UniDAC driver*
- for instance, you can set for both SQLite3 and Firebird/Interbase:
  
  Props.SpecificOptions.Values['ClientLibrary'] := ClientDllName;

TSQLDBUniDACConnection = class(TSQLDBConnectionThreadSafe)
  implements a direct connection via UniDAC database access

constructor Create(aProperties: TSQLDBConnectionProperties); override;
  *Prepare a connection for a specified UniDAC database access*

destructor Destroy; override;
  *Release memory and connection*

function IsConnected: boolean; override;
  *Return TRUE if Connect has been already successfully called*

function NewStatement: TSQLDBStatement; override;
  *Create a new statement instance*

procedure Commit; override;
  *Commit changes of a Transaction for this connection*
- StartTransaction method must have been called before

procedure Connect; override;
  *Connect to the specified database server using UniDAC*
- should raise an ESQLDBUniDAC on error

procedure Disconnect; override;
  *Stop connection to the specified database server using UniDAC*
- should raise an ESQLDBUniDAC on error

procedure Rollback; override;
  *Discard changes of a Transaction for this connection*
- StartTransaction method must have been called before

procedure StartTransaction; override;
  *Begin a Transaction for this connection*

property Database: TUniConnection read fDatabase;
  *Access to the associated UniDAC connection instance*
TSQLDBUniDACStatement = class(TSQLDBDatasetStatement)

    implements a statement via a UniDAC connection

Constants implemented in the SynDBUniDAC unit

UNIDAC_PROVIDER: array[doOracle..high(TSQLDBDefinition)] of RawUTF8 = ('Oracle','SQL Server','Access','MySQL','SQLite','InterBase','NexusDB','PostgreSQL','DB2','');

    UniDAC provider names corresponding to SynDB recognized SQL engines
28. SynFile application

This sample application is a simple database tool which stores text content and files into the database, in both clear and "safe" manner. Safe records are stored using AES/SHA 256-bit encryption. There is an Audit Trail table for tracking the changes made to the database.

This document will follow the application architecture and implementation, in order to introduce the reader to some main aspects of the Framework:
- General architecture - see Multi-tier architecture (page 88);
- Database design - see Object-Relational Mapping (ORM) (page 92);
- User Interface generation.

We hope this part of the Software Architecture Design (SAD) document will be able to be a reliable guideline for using our framework for your own projects.
28.1. General architecture

According to the Multi-tier architecture, some units will define the three layers of the SynFile application:

Database Model

First, the database tables are defined as regular Delphi classes, like a true ORM framework. Classes are translated to database tables. Published properties of these classes are translated to table fields. No external configuration files to write - only Delphi code. Nice and easy. See FileTables.pas unit.

This unit is shared by both client and server sides, with a shared data model, i.e. a TSQLModel class instance, describing all ORM tables/classes.

It contains also internal event descriptions, and actions, which will be used to describe the software UI.

Business Logic

The server side is defined in a dedicated class, which implements an automated Audit Trail, and a service named "Event" to easily populate the Audit Trail from the Client side. See FileServer.pas unit.

The client side is defined in another class, which is able to communicate with the server, and fill/update/delete/add the database content playing with classes instances. It's also used to call the Audit Trail related service, and create the reports. See FileClient.pas unit.

Client-Server logic will be detailed in the next paragraph.

Presentation Layer

The main form of the Client is void, if you open its FileMain.dfm file. All the User Interface is created by the framework, dynamically from the database model and some constant values and enumeration types (thanks to Delphi RTTI) as defined in FileTables.pas unit (the first one, which defines also the classes/tables).

It's main method is TMainForm.ActionClick, which will handle the actions, triggered when a button is pressed.

The reports use GDI+ for anti-aliased drawing, can be zoomed and saved as pdf or text files.

The last FileEdit.pas unit is just the form used for editing the data. It also performs the encryption of "safe memo" and "safe data" records, using our SynCrypto.pas unit. It will AES-NI hardware instructions, if available, so will be very fast, even for big content.

You'll discover how the ORM plays its role here: you change the data, just like changing any class instance properties.

It also uses our SynGdiPlus.pas unit to create thumbnails of any picture (emf+jpg+tif+gif+bmp) of data inserted in the database, and add a BLOB data field containing these thumbnails.
28.2. Database design

The FileTables.pas unit is implementing all TSQLRecord child classes, able to create the database tables, using the ORM aspect of the framework - see Object-Relational Mapping (ORM) (page 92). The following class hierarchy was designed:

![SynFile TSQLRecord classes hierarchy](image)

Most common published properties (i.e. Name, Created, Modified, Picture, KeyWords) are taken from the TSQLFile abstract parent class. It's called "abstract", not in the current Delphi OOP terms, but as a class with no "real" database table associated. It was used to define the properties only once, without the need of writing the private variables nor the getter/setter for children classes. Only TSQLAuditTrail won't inherit from this parent class, because it's purpose is not to contain data, but just some information.

The database itself will define TSQLAuditTrail, TSQLMemo, TSQLData, TSQLSafeMemo, and TSQLSafeData classes. They will be stored as AuditTrail, Memo, Data, SafeMemo and SafeData tables in the SQLite3 database (the table names are extract from the class name, trimming the left 'TSQL' characters).

Here is this common ancestor type declaration:

```pascal
TSQLFile = class(TSQLRecordSigned)
  public
    fName: RawUTF8;
    fModified: TTimeLog;
    fCreated: TTimeLog;
    fPicture: TSQLRawBlob;
    fKeyWords: RawUTF8;
  published
    property Name: RawUTF8 read fName write fName;
    property Created: TTimeLog read fCreated write fCreated;
    property Modified: TTimeLog read fModified write fModified;
    property Picture: TSQLRawBlob read fPicture write fPicture;
    property KeyWords: RawUTF8 read fKeyWords write fKeyWords;
end;
```

Sounds like a regular Delphi class, doesn't it? The only fact to be noticed is that it does not inherit from a TPersistent class, but from a TSQLRecord class, which is the parent object type to be used for our
ORM. The TSQLRecordSigned class type just defines some Signature and SignatureTime additional properties, which will be used here for handling digital signing of records.

Here follows the Delphi code written, and each corresponding database field layout of each registered class:

```delphi
TSQLMemo = class(TSQLFile)
  public
    fContent: RawUTF8;
  published
    property Content: RawUTF8 read fContent write fContent;
end;
```

**Memo Record Layout**

```
TSQLData = class(TSQLFile)
  public
    fData: TSQLRawBlob;
  published
    property Data: TSQLRawBlob read fData write fData;
end;
```

**Data Record Layout**

```
TSQLSafeMemo = class(TSQLData);
```

**SafeMemo Record Layout**

```
TSQLSafeData = class(TSQLData);
```
You can see that TSQLSafeMemo and TSQLSafeData are just a direct sub-class of TSQLData to create "SafeMemo" and "SafeData" tables with the exact same fields as the "Data" table. Since they were declared as class(TSQLData), they are some new class type.

Then the latest class is not inheriting from TSQLFile, because it does not contain any user data, and is used only as a log of all actions performed using SynFile:

```pascal
class TSQLAuditTrail = class(TSQLRecord)
  protected
    fStatusMessage: RawUTF8;
    fStatus: TFileEvent;
    fAssociatedRecord: TRecordReference;
    fTime: TTimeLog;
  published
    property Time: TTimeLog read fTime write fTime;
    property Status: TFileEvent read fStatus write fStatus;
    property StatusMessage: RawUTF8 read fStatusMessage write fStatusMessage;
    property AssociatedRecord: TRecordReference read fAssociatedRecord write fAssociatedRecord;
end;
```

The AssociatedRecord property was defined as TRecordReference. This special type (mapped as an INTEGER field in the database) is able to define a "one to many" relationship with ANY other record of the database model.

- If you want to create a "one to many" relationship with a particular table, you should define a property with the corresponding TSQLRecord sub-type (for instance, if you want to link to a particular SafeData row, define the property as AssociatedData: TSQLSafeData;) - in this case, this will create an INTEGER field in the database, holding the RowID value of the associated record (and this field content will be filled with pointer(RowID) and not with a real TSQLSafeData instance).

- Using a TRecordReference type won't link to a particular table, but any table of the database model: it will store in its associated INTEGER database field not only the RowID of the record, but also the table index as registered at TSQLModel creation. In order to access this AssociatedRecord property content, you could use either TSQLRest. Retrieve(AssociatedRecord) to get the corresponding record instance, or typecast it to RecordRef wrapper structure to easily retrieve or set the associated table and RowID. You could also use the TSQLRecord. RecordReference(Model) method in order to get the value corresponding to an existing TSQLRecord instance.
According to the MVC model - see Model-View-Controller (page 86) - the framework expect a common database model to be shared between client and server. A common function has been defined in the FileTables.pas unit, as such:

```pascal
function CreateFileModel(Owner: TSQLEst): TSQLModel;
```

We’ll see later its implementation. Just note for the moment that it will register the TSQLAuditTrail, TSQLMemo, TSQLData, TSQLSafeMemo, and TSQLSafeData classes as part of the database model. The order of the registration of those classes will be used for the AssociatedRecord::TRecordReference field of TSQLAuditTrail - e.g. a TSQLMemo record will be identified with a table index of 1 in the RecordReference encoded value. So it’s mandatory to NOT change this order in any future modification of the database schema, without providing any explicit database content conversion mechanism.

Note that all above graphs were created directly from our SynProject tool, which is able to create custom graphs from the application source code it parsed.
28.3. Client Server implementation

Server-side is implemented in unit FileServer, with the following class:

```pascal
TFileServer = class(TSQLRestserverDB)
  (...)
  Server: TSQLHttpServer;
  procedure AddAuditTrail(aEvent: TFileEvent; const aMessage: RawUTF8='';
        aAssociatedRecord: TRecordReference=0);
  function OnDatabaseUpdateEvent(Sender: TSQLRestServer;
        Event: TSQLEvent; aTable: TSQLRecordClass; aID: TID): boolean;
  published
  procedure Event(Ctxt: TSQLRestServerURIContext);
end;
```

As stated above, it inherits from TSQLRestserverDB to define a RESTful ORM based on the SQLite3 database engine, and define a custom method-based service named Event.

The class constructor creates the whole server-side logic, following the shared data Model as defined in the FileTables unit:

```pascal
constructor TFileServer.Create;
begin
  inherited Create(CreateFileModel(self),ChangeFileExt(paramstr(0),'.db3'));
  CreateMissingTables(ExeVersion.Version.Version32);
  Server := TSQLHttpServer.Create(SERVER_HTTP_PORT,self,'+',useHttpApiRegisteringURI);
  AddAuditTrail(feServerStarted);
  OnUpdateEvent := OnDatabaseUpdateEvent;
end;
```

A dedicated TSQLHttpServer instance is initialized to publish the TFileServer content over HTTP.

Automatic logging in the Audit Trail table of most database changes is performed via the OnDatabaseUpdateEvent callback.

Client-side is implemented in unit FileClient, with the following class:

```pascal
TFileClient = class(TSQLHttpClient)
  public
  (...)
  procedure AddAuditTrail(aEvent: TFileEvent; aAssociatedRecord: TSQLRecord);
end;
```

You'll see that BLOB fields are handled just like other fields, even if they use their own RESTful GET/PUT dedicated URI (they are not JSON encoded, but transmitted as raw data, to save bandwidth and maintain the RESTful model). The framework handles it for you, thanks to its ORM orientation, in the TFileClient constructor:

```pascal
constructor TFileClient.Create(const aServer: AnsiString);
begin
  inherited Create(aServer,SERVER_HTTP_PORT,CreateFileModel(self));
  ForceBlobTransfer := true;
end;
```

Here, we set ForceBlobTransfer := true, since by default all BLOB fields won't be transmitted by TSQLRestClientURI, whereas our simple application expect them to be always available.

The same data Model, as defined in the FileTables unit, is used also on the client side.

On both sides, a AddAuditTrail() is defined, to allow direct logging to the internal TSQLAuditTrail table. From the client, it uses the Event method-based service to perform the remote action:

```pascal
procedure TFileClient.AddAuditTrail(aEvent: TFileEvent;
        aAssociatedRecord: TSQLRecord);
```
begin
  if aAssociatedRecord=nil then
    CallBackGetResult('Event',['event',ord(aEvent)])
  else
    with aAssociatedRecord do
      CallBackGetResult('Event',['event',ord(aEvent)],RecordClass,ID);
  end;
28.4. User Interface generation

You could of course design your own User Interface without our framework. That is, this is perfectly feasible to use only the ORM part of it. For instance, it should be needed to develop AJAX applications using its RESTful model - see REST (page 311) - since such a feature is not yet integrated to our provided source code.

But for producing easily applications, the framework provides a mechanism based on both ORM description and RTTI compiler-generated information in order to create most User Interface by code.

It is able to generated a Ribbon-based application, in which each table is available via a Ribbon tab, and some actions performed to it.

So the framework will need to know:
- Which tables must be displayed;
- Which actions should be associated with each table;
- How the User Interface should be customized (e.g. hint texts, grid layout on screen, reporting etc...);
- How generic automated edition, using the mORMotUIEdit.pas unit, is to be generated.

To this list could be added an integrated event feature, which can be linked to actions and custom status, to provide a centralized handling of user-level logging (as used e.g. in the SynFile TSQLAuditTrail table) - please do not make confusion between this user-level logging and technical-level logging using TSynLog and TSQLLog classes and "families" - see Enhanced logging (page 631).

28.4.1. Rendering

The current implementation of the framework User Interface generation handles two kind of rendering:
- Native VCL components;
- Proprietary TMS components.

You can select which set of components are used, by defining - globally to your project (i.e. in the Project/Options/Conditionals menu) - the USETMSPACK conditional. If it is not set (which is by default), it will use VCL components.

The native VCL components will use native Windows API components. So the look and feel of the application will vary depending on the Windows version it is running on. For instance, the resulting screen will be diverse if the application is run under Windows 2000, XP, Vista and Seven. The "ribbon" as generated with VCL components has most functionalities than the Office 2007/2010 ribbon, but will have a very diverse layout.

The TMS components will have the same rendering whatever the Windows it's running on, and will display a "ribbon" very close to the official Office 2007/2010 version.

Here are some PROs and CONs about both solutions:

<table>
<thead>
<tr>
<th>Criteria</th>
<th>VCL</th>
<th>TMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rendering</td>
<td>Basic</td>
<td>Sophisticated</td>
</tr>
<tr>
<td>OS version</td>
<td>Variant</td>
<td>Constant</td>
</tr>
</tbody>
</table>
Ribbon look | Unusual | Office-like
--- | --- | ---
Preview button & Shortcuts | None by default | Available
Extra Price | None | High
GPL ready | Yes | No
Office UI Licensing | N/A | Required
EXE size | Smaller | Bigger

It's worth saying that the choice of one or other component set could be changed on request. If you use the generic components as defined in mORMotToolBar (i.e. the TSynForm, TSynToolBar, TSynToolBarButton, TSynPopupMenu, TSynPage, TSynPager, TSynBodyPager and TSynBodyPage classes) and SynTaskDialog (for TSynButton) in your own code, the USETMSPACK conditional will do all the magic for you.

The Office UI licensing program was designed by Microsoft for software developers who wish to implement the Office UI as a software component and/or incorporate the Office UI into their own applications. If you use TMS ribbon, it does not require any more acceptance of the Office UI License terms - see at http://msdn.microsoft.com/en-us/office/aa973809.aspx.


Here is the screen content, using the TMS components:

And here is the same application compiled using only VCL components, available from Delphi 6 up to Delphi 10.3 Rio:
We did not use yet the Ribbon component as was introduced in Delphi 2009. Its action-driven design won't make it easy to interface with the event-driven design of our User Interface handling, and we have to confess that this component has rather bad reputation (at least in the Delphi 2009 version). Feel free to adapt our Open Source code to use it - we'll be very pleased to release a new version supporting it, but we don't have time nor necessity to do it by ourself.

28.4.2. Enumeration types

A list of available actions should be defined, as an enumeration type:

```pascal
TFileAction = (faNoAction, faMark, faUnmarkAll, faQuery, faRefresh, faCreate, faEdit, faCopy, faExport, faImport, faDelete, faSign, faPrintPreview, faExtract, faSettings);
```

Thanks to the Delphi RTTI, and "Un Camel Casing", the following list will generate a set of available buttons on the User Interface, named "Mark", "Unmark all", "Query", "Refresh", "Create", "Edit", "Copy", "Export", "Import", "Delete", "Sign", "Print preview", "Extract" and "Settings". Thanks to the mORMoti18n.pas unit (responsible of application i18n) and the TLanguageFile. Translate method, it could be translated on-the-fly from English into the current desired language, before display on screen or report creation.

See both above screen-shots to guess how the button captions match the enumeration names - i.e. User Interface generated using VCL components (page 2515) and User Interface generated using VCL components (page 2515).

A list of events, as used for the TSQLAuditTrail table, was also defined. Some events reflect the change made to the database rows (like feRecordModified), or generic application status (like feServerStarted):
In the grid and the reports, RTTI and "uncamelcasing" will be used to display this list as regular text, like "Record digitally signed", and translated to the current language, if necessary.

28.4.3. ORM Registration

The User Interface generation will be made by creating an array of objects inheriting from the TSQLRibbonTabParameters type.

Firstly, a custom object type is defined, associating

```pascal
TFileRibbonTabParameters = object(TSQLRibbonTabParameters)
  /// the SynFile actions
  Actions: TFileActions;
end;
```

Then a constant array of such objects is defined:

```pascal
const
FileTabs: array[0..4] of TFileRibbonTabParameters = (
  (Table: TSQLAuditTrail;
   Select: 'Time,Status,StatusMessage'; Group: GROUP_MAIN;
   FieldWidth: 'gIZ'; ShowID: true; ReverseOrder: true; Layout: llClient;
   Actions: [faDelete,faMark,faUnmarkAll,faQuery,faRefresh,faPrintPreview,faSettings]),
  (Table: TSQLMemo;
   Select: DEF_SELECT; Group: GROUP_CLEAR; FieldWidth: 'IddId'; Actions: DEF_ACTIONS),
  (Table: TSQLDataSource;
   Select: DEF_SELECT; Group: GROUP_CLEAR; FieldWidth: 'IddId'; Actions: DEF_ACTIONS_DATA),
  (Table: TSQLSafeMemo;
   Select: DEF_SELECT; Group: GROUP_SAFE; FieldWidth: 'IddId'; Actions: DEF_ACTIONS),
  (Table: TSQLSafeData;
   Select: DEF_SELECT; Group: GROUP_SAFE; FieldWidth: 'IddId'; Actions: DEF_ACTIONS_DATA));
```

The Table property will map the ORM class to the User Interface ribbon tab. A custom CSV list of fields should be set to detail which database columns must be displayed on the grids and the reports, in the Select property. Each ribbon tab could contain one or more TSQLRecord table: the Group property is set to identify on which ribbon group it should be shown. The grid column widths are defined as a FieldWidth string in which each displayed field length mean is set with one char per field (A=first Select column,Z=26th column) - lowercase character will center the field data. For each table, the available actions are also set, and will be used to create the possible buttons to be shown on the ribbon toolbars (enabling or disabling a button is to be done at runtime).

Note that this array definition uses some previously defined individual constants (like DEF_SELECT, DEF_ACTIONS_DATA or GROUP_SAFE. This is a good practice, and could make code maintenance easier later on.

28.4.4. Main window

Once all this ORM and action information is available, the FileMain unit defines the following class to generate the expected ribbon-based User Interface:

```pascal
TMainForm = class(TSynForm)
  ImageList32: TImageList;
  ImageList16: TImageList;
  procedure FormCreate(Sender: TObject);
  procedure FormShow(Sender: TObject);
private
{$ifdef DEBUGINTERNALSERVER}
procedure TFormCreate(Sender: TObject);
var P: integer;
begin
{$ifdef DEBUGINTERNALSERVER}
try
  Server := TFileServer.Create;
except
  on E: Exception do begin
    ShowException(E);
    exit;
  end;
end;
{$endif}
LoadImageListFromEmbeddedZip(ImageList32, 'buttons.bmp');
ImageListStretch(ImageList32, ImageList16);
Client := TFileClient.Create('localhost');
Client.OnIdle := TLoginForm.OnIdleProcessForm;
Ribbon := TFileRibbon.Create(self, nil, nil, ImageList32, ImageList16,
  Client, ALL_ACCESS_RIGHTS, nil, Client.OnSetAction, sFileActionsToolbar,
  sFileActionsHints, nil, ActionClick, integer(faRefresh), 1, false,
  length(FileTabs), @FileTabs[0], sizeof(FileTabs[0]),
  sFileTabsGroup, ',BannerData,BannerSafe',true);
Ribbon.ToolBar.Caption.Caption := Caption;
Ribbon.ToolBar.HelpButton.OnClick := HelpClick;
for P := 0 to high(Ribbon.Page) do
  with Ribbon.Page[P] do
    if Lister<>nil then
      Lister.Grid.OnDb1Click := ListDb1Click;
end;

Even if a real application may be truly Client-Server, we define a stand-alone mode. That is, a TFileServer instance is instantiated within the main application execution. Just un-define the DEBUGINTERNALSERVER conditional if you want a "pure client" version of the application - in this case, a stand-alone server shall be running.

All the ORM and actions defined in the FileTables unit are used to initialize the TFileRibbon content in the Ribbon field which will be the main entry point of all User Interface process.

The ActionClick() method is the main entry point of the application, and is called when the User clicks on any ribbon button. It is just a case Action of ... switch instruction, handling each TFileAction event as expected.

The Edit() method will allow edition of a given record fields, via the separated TEditForm window, as defined in FileEdit unit. We won't use the auto-generated window from RTTI in this case, since we expect a dedicated process to attach a picture to the corresponding TSQLFile item.

The ListDb1Click() method will process any double click on the list to edit the corresponding item (faEdit action), or go to the record an audit trail row refers to, using a convenient local RecordRef
wrapper variable:

```pascal
procedure TMainForm.ListDblClick(Sender: TObject);
var P: TSQLRibbonTab;
  ref: RecordRef;
begin
  P := Ribbon.GetActivePage;
  if P<>nil then
    if P.Table=TSQLAuditTrail then begin
      if P.Retrieve(Client,P.List.Row) then begin
        ref.Value := TSQLAuditTrail(P.CurrentRecord).AssociatedRecord;
        Ribbon.GotoRecord(ref.Table(Client.Model),ref.ID);
      end;
    end else
    ActionClick(Sender,P.Table,ord(faEdit));
end;
```

The `WMRefreshTimer()` method will just transmit any `WM_TIMER` event to the ribbon process, in order to handle automatic refresh of the content, following the *stateless* approach of our RESTful framework:

```pascal
procedure TMainForm.WMRefreshTimer(var Msg: TWMTimer);
begin
  Ribbon.WMRefreshTimer(Msg);
end;
```

You probably noticed that `Client.OnIdle` is set in the `FormCreate` method to map the `TLoginForm.OnIdleProcessForm` callback. This will let the HTTP client class to use a background thread for all communication, instead of blocking the main application thread. The main User Interface will still be responsive (since `OnIdleProcessForm` will call `Application.ProcessMessages`), and change the cursor to `crHourGlass` in case of a slow request, or even display a temporary pop-up with "Please wait..." if the network is really slow, and the request takes more than 2 seconds (all those notification parameters can be changed in `mORMotUILogin.pas`).
28.5. Report generation

The following CreateReport method is overridden in FileClient.pas:

```pascal
/// class used to create the User interface
TFileRibbon = class(TSQLRibbon)
public
  /// overridden method used customize the report content
  procedure CreateReport(aTable: TSQLRecordClass; aID: TID; aReport: TGDIPages;
   AlreadyBegan: boolean=false); override;
end;
```

The reporting engine in the framework is implemented via the TGDIPages class, defined in the mORMotReport.pas:

- Data is drawn in memory, they displayed or printed as desired;
- High-level reporting methods are available (implementing tables, columns, titles and such), but you can have access to a TCanvas property which allows any possible content generation via standard VCL methods;
- Allow preview (with anti-aliased drawing via GDI+) and printing;
- Direct export as .txt or .pdf file;
- Handle bookmark, outlines and links inside the document.

By default, the CreateReport method of TSQLRibbon will write all editable fields value to the content.

The method is overridden by the following code:

```pascal
procedure TFileRibbon.CreateReport(aTable: TSQLRecordClass; aID: TID; aReport: TGDIPages;
   AlreadyBegan: boolean=false);
var
  Rec: TSQLFile;
  Pic: TBitmap;
  s: string;
  PC: PChar;
  P: TSQLRibbonTab;
begin
  with aReport do
    begin
      // initialize report
      Clear;
      BeginDoc;
      Font.Size := 10;
      if not aTable.InheritsFrom(TSQLFile) then
        P := nil else
        P := GetActivePage;
      if (P=nil) or (P.CurrentRecord.ID<>aID) or (P.Table<>aTable) then
        inherited; // default handler
        exit;
      end;
      Rec := TSQLFile(P.CurrentRecord);
      Caption := U2S(Rec.fName);
      // prepare page footer
      SaveLayout;
      Font.Size := 9;
      AddPagesToFooterAt(sPageN,leftMargin);
      TextAlign := taRight;
      AddTextToFooterAt('SynFile  https://synopse.info - '+Caption,RightMarginPos);
```

The report is cleared, and BeginDoc method is called to start creating the internal canvas and band positioning. The font size is set, and parameters are checked against expected values. Then the current viewed record is retrieved from GetActivePage. CurrentRecord, and the report caption is set via the record Name field.
Page footer are set by using two methods:
- AddPagesToFooterAt to add the current page number at a given position (here the left margin);
- AddTextToFooterAt to add some custom text at a given position (here the right margin, after having changed the text alignment into right-aligned).

Note that SaveLayout/RestoreSavedLayout methods are used to modify temporary the current font and paragraph settings for printing the footer, then restore the default settings.

```delphi
// write global header at the beginning of the report
DrawTitle(P.Table.CaptionName+': '+'true);
NewHalfLine;
AddColumns([6,40]);
SetColumnBold(0);
if Rec.SignatureTime<>0 then
begin
  PC := Pointer(Format(sSignedN,[Rec.SignedBy,Iso2S(Rec.SignatureTime)]));
  DrawTextAcrossColsFromCSV(PC,$C0C0FF);
end;
if Rec.fCreated<>0 then
  DrawTextAcrossCols([sCreated,Iso2S(Rec.fCreated)]);
if Rec.fModified<>0 then
  DrawTextAcrossCols([sModified,Iso2S(Rec.fModified)]);
if Rec.fKeyWords='' then
  begin
    s := sNone
    begin
      s := U2S(Rec.fKeyWords);
      ExportPDFKeywords := s;
    end;
  end;
DrawTextAcrossCols([sKeyWords,s]);
NewLine;
Pic := LoadFromRawByteString(Rec.fPicture);
if Pic<>nil then
try
  DrawBMP(Pic,0,Pic.Width div 3);
finally
  Pic.Free;
end;
end;
```

Report header is written using the following methods:
- DrawTitle to add a title to the report, with a black line below it (second parameter to true) - this title will be added to the report global outline, and will be exported as such in .pdf on request;
- NewHalfLine and NewLine will leave some vertical gap between two paragraphs;
- AddColumns, with parameters set as percentages, will initialize a table with the first column content defined as bold (SetColumnBold(0));
- DrawTextAcrossCols and DrawTextAcrossColsFromCSV will fill a table row according to the text specified, one string per column;
- DrawBMP will draw a bitmap to the report, which content is loaded using the generic LoadFromRawByteString function implemented in SynGdiPlus.pas;
- U2S and Iso2S function, as defined in mORMoti18n.pas, are used for conversion of some text or TTimeLog/TUnixTime into a text formatted with the current language settings (i18n).

```delphi
// write report content
DrawTitle(sContent,true);
SaveLayout;
Font.Name := 'Courier New';
if Rec.InheritsFrom(TSQLSafeMemo) then
  DrawText(sSafeMemoContent) else
if Rec.InheritsFrom(TSQLMemo) then
  DrawTextU(TSQLMemo(Rec).Content) else
if Rec.InheritsFrom(TSQLData) then
  with TSQLData(Rec) do
```

```
begin
  DrawTextU(Rec.fName);
  s := PictureName(TSynPicture.IsPicture(TFileName(Rec.fName)));
  if s<>'' then
    s := format(sPictureN,[s]) else
  if not Rec.InheritsFrom(TSQLSafeData) then
    s := U2S(GetMimeContentType(Pointer(Data),Length(Data),TFileName(Rec.fName)));
  if s<>'' then
    DrawTextFmt(sContentTypeN,[s]);
    DrawTextFmt(sSizeN,[U2S(KB(Length(Data)))]);
    NewHalfLine;
    DrawText(sDataContent);
end;
RestoreSavedLayout;

Then the report content is appended, according to the record class type:
- DrawText, DrawTextU and DrawTextFmt are able to add a paragraph of text to the report, with the current alignment - in this case, the font is set to 'Courier New' so that it will be displayed with fixed width;
- GetMimeContentType is used to retrieve the exact type of the data stored in this record.

// set custom report parameters
ExportPDFApplication := 'SynFile https://synopse.info';
ExportPDFForceJPEGCompression := 80;
end;
end;

Those ExportPDFApplication and ExportPDFForceJPEGCompression properties (together with the ExportPDFKeywords are able to customize how the report will be exported into a .pdf file. In our case, we want to notify that SynFile generated those files, and that the header bitmap should be compressed as JPEG before writing to the file (in order to produce a small sized .pdf).

You perhaps did notice that textual constant were defined as resourcestring, as such:

resourcestring
  sCreated = 'Created';
  sModified = 'Modified';
  sKeyWords = 'KeyWords';
  sContent = 'Content';
  sNone = 'None';
  sPageN = 'Page %d / %d';
  sSizeN = 'Size: %s';
  sContentTypeN = 'Content Type: %s';
  sSafeMemoContent = 'This memo is password protected.'#13
    'Please click on the "Edit" button to show its content.';
  sDataContent = 'Please click on the "Extract" button to get its content.';
  sSignedN = 'Signed, By %s on %s';
  sPictureN = '%s Picture';

The mORMot18n.pas unit is able to parse all those resourcestring from a running executable, via its ExtractAllResources function, and create a reference text file to be translated into any handled language.

Creating a report from code does make sense in an ORM. Since we have most useful data at hand as Delphi classes, code can be shared among all kind of reports, and a few lines of code is able to produce complex reports, with enhanced rendering, unified layout, direct internationalization and export capabilities.

Note that the mORMotReport.pas unit uses UTF-16 encoded string, i.e. our SynUnicode type, which is either UnicodeString since Delphi 2009, or WideString for older versions. WideString is known to have performance issues, due to use of slow BSTR API calls - so if you want to create huge reports with pre-Unicode versions of Delphi and our report engine, consider adding a reference to our
SynFastWideString.pas unit at first place of your .dpr uses clause, for potential huge speed enhancement. See *Unicode and UTF-8* (page 105) for more details, especially the restriction of use, since it will break any attempt to use BSTR parameters with any OLE/COM object.
28.6. Application i18n and L10n

In computing, internationalization and localization (also spelled internationalisation and localisation) are means of adapting computer software to different languages, regional differences and technical requirements of a target market:

- **Internationalization** (i18n) is the process of designing a software application so that it can be adapted to various languages;
- **Localization** (L10n) is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text, e.g. for dates display.

Our framework handles both features, via the mORMot18n.pas unit. We just saw above how `resourcestring` defined in the source code are retrieved from the executable and can be translated on the fly. The unit extends this to visual forms, and even captions generated from RTTI - see **RTTI** (page 503).

The unit expects all textual content (both `resourcestring` and RTTI derived captions) to be correct English text. A list of all used textual elements will be retrieved then hashed into an unique numerical value. When a specific locale is set for the application, the unit will search for a `.msg` text file in the executable folder matching the expected locale definition. For instance, it will search for `FR.msg` for translation into French.

In order to translate all the user interface, a corresponding `.msg` file is to be supplied in the executable folder. Neither the source code, nor the executable is to be rebuild to add a new language. And since this file is indeed a plain textual file, even a non developer (e.g. an end-user) is able to add a new language, starting from another `.msg`.

28.6.1. Creating the reference file

In order to begin a translation task, the mORMot18n.pas unit is able to extract all textual resource from the executable, and create a reference text file, containing all English sentences and words to be translated, associated with their numerical hash value.

It will in fact:

- Extract all `resourcestring` text;
- Extract all captions generated from RTTI (e.g. from enumerations or class properties names);
- Extract all embedded dfm resources, and create per-form sections, allowing a custom translation of displayed captions or hints.

This creation step needs a compilation of the executable with the **EXTRACTALLRESOURCES** conditional defined, **globally** to the whole application (a full **rebuild** is necessary after having added or suppressed this conditional from the **Project / Options / Folders-Conditionals** IDE field).

Then the **ExtractAllResources** global procedure is to be called somewhere in the code.

For instance, here is how this is implemented in FileMain.pas, for the framework main demo:

```pascal
procedure TMainForm.FormShow(Sender: TObject);
begin
  {$ifdef EXTRACTALLRESOURCES}
  ExtractAllResources;
  
  // first, all enumerations to be translated
  [TypeInfo(TFileEvent),TypeInfo(TFileAction),TypeInfo(TPreviewAction)],
  
  // then some class instances (including the TSQLModel will handle all TSQLRecord)
  [Client.Model],
  
  // some custom classes or captions
```
A global EXTRACTALLRESOURCES conditional can be defined temporarily for the project: from the Delphi IDE, Project/Options then enabling the conditional, Project/Run to create the .messages file as expected, and finally Project/Options to undefine the EXTRACTALLRESOURCES conditional and rebuild a regular executable.

The TFileEvent and TFileAction enumerations RTTI information is supplied, together with the current TSQLModel instance. All TSQLRecord classes (and therefore properties) will be scanned, and all needed English caption text will be extracted.

The Close method is then called, since we don't want to use the application itself, but only extract all resources from the executable.

Running once the executable will create a SynFile.messages text file in the SynFile.exe folder, containing all English text:

```
[TEditForm]
Name.EditLabel.Caption=_2817614158   Name
KeyWords.EditLabel.Caption=_3731019706   KeyWords

[TLoginForm]
Label1.Caption=_1741937413   &User name:
Label2.Caption=_4235002365   &Password:

[TMainForm]
Caption=_16479868    Synopse mORMot demo

[Messages]
2784453965=Memo
2751226180=Data
744738530=Safe memo
895337940=Safe data
2817614158=Name
1741937413=&User name:
4235002365=&Password:
16479868= Synopse mORMot demo - SynFile
940170664=Content
3153227598=None
3708724895=Page %d / %d
2767358349=Size: %s
4281038646=Content Type: %s
2584741026=This memo is password protected.|Please click on the "Edit" button to show its content.
3011148197=Please click on the "Extract" button to get its content.
388288630=Signed,By %s on %s
(...)
```

The main section of this text file is named [Messages]. In fact, it contains all English extracted texts, as Numerical Key=English Text pairs. Note this will reflect the exact content of resource string or RTTI captions, including formatting characters (like %d), and replacing line feeds (#13) by the special \ character (a line feed is not expected on a one-line-per-pair file layout). Some other text lines are separated by a comma. This is usual for instance for hint values, as expected by the code.

As requested, each application form has its own section (e.g. [TEditForm], [TMainForm]), proposing some default translation, specified by a numerical key (for instance Label1.Caption will use the text identified by 1741937413 in the [Messages] section). The underline character before the numerical...
key is used to refer to this value. Note that if no _NumericalKey is specified, a plain text can be specified, in order to reflect a specific use of the generic text on the screen.

### 28.6.2. Adding a new language

In order to translate the whole application into French, the following FR.msg file could be made available in the SynFile.exe folder:

```plaintext
[Messages]
2784453965=Texte
downloaded the Synopse mORMot Framework demo - SynFile.exe
940170664=Contenu
devices a plain text can be specified, in order to reflect a specific use of the generic text on the screen.

Since no form-level custom captions (e.g. [TLoginForm]) have been defined in this FR.msg file, the default numerical values will be used. In our case, Name.EditLabel.Caption will be displayed using the text specified by 2817614158, i.e. 'Nom'. You can specify a custom translation for a given field on any form: sometimes, the text should be adapted with a given context.

Note that the special characters %s %d , | markup was preserved: only the plain English text has been translated to the corresponding French.

### 28.6.3. Language selection

User Interface language can be specified at execution.

There are two ways to change the application language:
- Manual translation of every form;
- Hook of the common TForm / TFrame classes, for automatic translation.

In manual translation mode:
- You can change languages on the fly, i.e. no need to restart the application;
- But you must modify your code to explicitly translate the forms after their creation;
- And you won't be able to translate dialogs without sources (e.g. third-party dialogs).

TForm/TFrame hook, on its side, has the following behavior:
- You do not need to modify your code, since it will be global to the application;
- It will work also for any third-party dialog, even if you do not have the source of it;
- But you can't change the language on the fly: you need to restart the application.

### 28.6.4. Manual translation

Once for the application, you should call SetCurrentLanguage() to set the global Language object and all related Delphi locale settings.
The, in each OnShow event of any form, you should call FormTranslateOne() e.g.

```delphi
procedure TMyForm.FormShow(Sender: TObject);
begin
    Language.FormTranslateOne(self);
end;
```

Another possibility may be to translate all already allocated forms at once, e.g. in the OnShow event of the application's main form:

```delphi
Language.FormTranslate([MainForm, FormTwo, FormAbout]);
```

Note that a list of already translated forms is maintained by the unit, when you call FormTranslate(). Therefore:

- All specified forms will be translated again by any further SetCurrentLanguage() call;
- But none of these forms must be freed after a FormTranslate([]) call - use FormTranslateOne() instead to translate a given form once, e.g. for all temporary created forms.

### 28.6.5. TForm / TFrame hook

If the USEFORMCREATEHOOK conditional is defined, the mORMoti18n.pas unit will hook TCustomForm.OnCreate method to translate all its nested components. It will also intercept TCustomFrame.Create() to allow automatic translation of its content.

Since the language must be known at program startup, before any TForm is actually created, the language will be set in the Operating System registry. The `HKEY_CURRENT_USER\Software\[CompanyName]i18n` key should contain one value per application (i.e. the lowercase .exe file name without its path), which will identify the abbreviation of the expected language. If there is no entry in this registration key for the given application, the current Windows local will be used.

For instance, if you define USEFORMCREATEHOOK conditional for your project, and run at least e.g. once in FileMain.pas, for the framework main demo:

```delphi
i18nLanguageToRegistry(lngFrench);
```

.. then it will set the main application language as French. At next startup, the unit will search for a FR.msg file, which will be used to translate all screen layout, including all RTTI-generated captions.

Of course, for a final application, you'll need to change the language by a common setting. See `i18nAddLanguageItems`, `i18nAddLanguageMenu` and `i18nAddLanguageCombo` functions and procedures to create your own language selection dialog, using a menu or a combo box, for instance.

### 28.6.6. Localization

Take a look at the TLanguageFile class. After the main language has been set, you can use the global Language instance in order to localize your application layout.

The mORMoti18n unit will register itself to some methods of mORMot.pas, in order to translate the RTTI-level text into the current selected language. See for instance `i18nDateText`. 
29. Main SynFile Demo source

29.1. Main SynFile Demo used Units
The Main SynFile Demo makes use of the following units.

**Units located in the "Lib" directory:**

<table>
<thead>
<tr>
<th>Source File Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
<tr>
<td>SynGdiPlus</td>
<td>GDI+ library API access</td>
<td>1350</td>
</tr>
<tr>
<td>SynTable</td>
<td>Filter/database/cache/buffer/security/search/multithread/OS features</td>
<td>1721</td>
</tr>
<tr>
<td>SynTaskDialog</td>
<td>Implement TaskDialog window (native on Vista/Seven, emulated on XP)</td>
<td>1824</td>
</tr>
<tr>
<td>SynZip</td>
<td>Low-level access to ZLib compression (1.2.5 engine version)</td>
<td>1845</td>
</tr>
</tbody>
</table>

---

**Unit dependencies in the "Lib" directory**

---

**Units located in the "Lib\SQLite3" directory:**

<table>
<thead>
<tr>
<th>Source File Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td>mORMotHttpClient</td>
<td>HTTP/1.1 RESTful JSON Client classes for mORMot</td>
<td>2306</td>
</tr>
<tr>
<td>mORMoti18n</td>
<td>Internationalization (i18n) routines and classes for mORMot</td>
<td>2322</td>
</tr>
</tbody>
</table>
### Source File Name

<table>
<thead>
<tr>
<th>Source File Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mORMotReport</code></td>
<td>Reporting unit</td>
<td>2352</td>
</tr>
<tr>
<td><code>mORMotToolBar</code></td>
<td>ORM-driven Office 2007 Toolbar for mORMot</td>
<td>2390</td>
</tr>
<tr>
<td><code>mORMotUI</code></td>
<td>Grid to display database content for mORMot</td>
<td>2404</td>
</tr>
<tr>
<td><code>mORMotUILogin</code></td>
<td>Some common User Interface functions and dialogs for mORMot</td>
<td>2418</td>
</tr>
</tbody>
</table>

### Unit dependencies in the "Lib\SQLite3" directory

Units located in the "Lib\SQLite3\Samples\MainDemo" directory:

<table>
<thead>
<tr>
<th>Source File Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FileClient</code></td>
<td>SynFile client handling</td>
<td>2530</td>
</tr>
<tr>
<td><code>FileEdit</code></td>
<td>SynFile Edit window</td>
<td>2532</td>
</tr>
<tr>
<td><code>FileMain</code></td>
<td>SynFile main Window</td>
<td>2534</td>
</tr>
<tr>
<td><code>FileServer</code></td>
<td>SynFile server handling</td>
<td>2536</td>
</tr>
<tr>
<td><code>FileTables</code></td>
<td>SynFile ORM definitions shared by both client and server</td>
<td>2538</td>
</tr>
</tbody>
</table>
Unit dependencies in the "Lib\SQLite3\Samples\MainDemo" directory
29.2. FileClient.pas unit

*Purpose:* SynFile client handling

**Units used in the FileClient unit**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileTables</td>
<td>SynFile ORM definitions shared by both client and server</td>
<td>2538</td>
</tr>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMotHttpClient</td>
<td>HTTP/1.1 RESTful JSON Client classes for mORMot</td>
<td>2306</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMoti18n</td>
<td>Internationalization (i18n) routines and classes for mORMot</td>
<td>2322</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMotReport</td>
<td>Reporting unit</td>
<td>2352</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>mORMotToolBar</td>
<td>ORM-driven Office 2007 Toolbar for mORMot</td>
<td>2390</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td>SynGdiPlus</td>
<td>GDI+ library API access</td>
<td>1350</td>
</tr>
<tr>
<td></td>
<td>- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- make available most useful GDI+ drawing methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- allows AntiAliased rendering of any EMF file using GDI+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

**FileClient class hierarchy**

**Objects implemented in the FileClient unit**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFileClient</td>
<td>A HTTP/1.1 client to access SynFile</td>
<td>2531</td>
</tr>
</tbody>
</table>
Objects | Description | Page
--- | --- | ---
TFileVersion | Class used to create the User interface | 2531

**TFileVersion = class(TSQLHttpClient)**

*A HTTP/1.1 client to access SynFile*

**constructor** Create(const aServer: AnsiString); **reintroduce**;

*Initialize the Client for a specified network Server name*

**function** OnSetAction(TableIndex, ToolbarIndex: integer; TestEnabled: boolean; var Action): string;

*Used internally to retrieve a given action*

**procedure** AddAuditTrail(aEvent: TFileEvent; aAssociatedRecord: TSQLRecord);

*Client-side access to the remote RESTful service*

**TFileVersion = class(TSQLRibbon)**

*Class used to create the User interface*

**procedure** CreateReport(aTable: TSQLRecordClass; aID: TID; aReport: TGDPages; AlreadyBegun: boolean=false); **override**;

*Overridden method used customize the report content*
### 29.3. `FileEdit.pas` unit

**Purpose:** SynFile Edit window

#### Units used in the `FileEdit` unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FileTables</code></td>
<td>SynFile ORM definitions shared by both client and server</td>
<td>2538</td>
</tr>
<tr>
<td><code>mORMot</code></td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><code>mORMoti18n</code></td>
<td>Internationalization (i18n) routines and classes for mORMot</td>
<td>2322</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><code>mORMotUI</code></td>
<td>Grid to display database content for mORMot</td>
<td>2404</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><code>mORMotUILogin</code></td>
<td>Some common User Interface functions and dialogs for mORMot</td>
<td>2418</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><code>SynCommons</code></td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><code>SynCrypto</code></td>
<td>Fast cryptographic routines (hashing and cypher)</td>
<td>1139</td>
</tr>
<tr>
<td></td>
<td>- implements AES,XOR,ADLER32,MDS,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><code>SynGdiPlus</code></td>
<td>GDI+ library API access</td>
<td>1350</td>
</tr>
<tr>
<td></td>
<td>- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- make available most useful GDI+ drawing methods</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- allows Antiialiased rending of any EMF file using GDI+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
<tr>
<td><code>SynTaskDialog</code></td>
<td>Implement TaskDialog window (native on Vista/Seven, emulated on XP)</td>
<td>1824</td>
</tr>
<tr>
<td></td>
<td>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td></td>
</tr>
</tbody>
</table>

`FileEdit` class hierarchy

```plaintext
TVistaForm -> TEditForm
```
Objects implemented in the *FileEdit* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEditForm</td>
<td>SynFile Edit window</td>
<td>2533</td>
</tr>
</tbody>
</table>

TEditForm = class(TVistaForm)

*SynFile Edit window*

- we don't use the standard Window generation (from mORMotUIEdit), but a custom window, created as RAD

  function LoadPicture(const FileName: TFileName; var Picture: RawByteString): boolean;

  *Used to load a picture file into a BLOB content after 80% JPEG compression*

  function SetRec(const Value: TSQLFile): boolean;

  *Set the associated record to be edited*

  property ReadOnly: boolean read fReadOnly write fReadOnly;

  *Should be set to TRUE to disable any content editing*

  property Rec: TSQLFile read fRec;

  *Read-only access to the edited record*

Functions or procedures implemented in the *FileEdit* unit

<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cypher</td>
<td>Will display a modal form asking for a password, then encrypt or uncrypt some BLOB content</td>
<td>2533</td>
</tr>
</tbody>
</table>

  function Cypher(const Title: string; var Content: TSQLRawBlob; Encrypt: boolean): boolean;

  *Will display a modal form asking for a password, then encrypt or uncrypt some BLOB content*

  - returns TRUE if the password was correct and the data processed

  - returns FALSE on error (canceled or wrong password)

Variables implemented in the *FileEdit* unit

<table>
<thead>
<tr>
<th>EditForm: TEditForm;</th>
</tr>
</thead>
</table>

  *SynFile Edit window instance*
## 29.4. FileMain.pas unit

*Purpose:* SynFile main Window

### Units used in the FileMain unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileClient</td>
<td>SynFile client handling</td>
<td>2530</td>
</tr>
<tr>
<td>FileEdit</td>
<td>SynFile Edit window</td>
<td>2532</td>
</tr>
<tr>
<td>FileServer</td>
<td>SynFile server handling</td>
<td>2536</td>
</tr>
<tr>
<td>FileTables</td>
<td>SynFile ORM definitions shared by both client and server</td>
<td>2538</td>
</tr>
</tbody>
</table>
| mORMot               | Common ORM and SOA classes for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                             | 1899 |
| mORMotHttpClient     | HTTP/1.1 RESTful JSON Client classes for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                | 2306 |
| mORMoti18n           | Internationalization (i18n) routines and classes for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                              | 2322 |
| mORMotToolBar        | ORM-driven Office 2007 Toolbar for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                 | 2390 |
| mORMotUI             | Grid to display database content for mORMot
- this unit is a part of the freeware mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                     | 2404 |
| mORMotUILogin        | Some common User Interface functions and dialogs for mORMot
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                | 2418 |
| SynCommons           | Common functions used by most Synopse projects
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                 | 717  |
| SynGdiPlus           | GDI+ library API access
- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic
- make available most useful GDI+ drawing methods
- allows Antialiased rendering of any EMF file using GDI+
- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18 | 1350 |
| SynTable             | Filter/database/cache/buffer/security/search/multithread/OS features
- as a complement to SynCommons, which tended to increase too much
- licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                     | 1721 |
Objects implemented in the *FileMain* unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMainForm</td>
<td>SynFile main Window</td>
<td>2535</td>
</tr>
</tbody>
</table>

TMainForm = class(TSynForm)

*SynFile main Window*

Client: TFileClient;
*The associated database client*

Ribbon: TFileRibbon;
*The associated Ribbon which will handle all User Interface*

destructor Destroy; *override;*
*Release all used memory*
29.5. FileServer.pas unit

Purpose: SynFile server handling

Units used in the FileServer unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileTables</td>
<td>SynFile ORM definitions shared by both client and server</td>
<td>2538</td>
</tr>
</tbody>
</table>
| mORMot      | Common ORM and SOA classes for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                                                                                                                                 | 1899 |
| mORMoti18n  | Internationalization (i18n) routines and classes for mORMot  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                                                                                                                                 | 2322 |
| SynCommons  | Common functions used by most Synopse projects  
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                                                                                                                                  | 717  |
| SynTable    | Filter/database/cache/buffer/security/search/multithread/OS features  
- as a complement to SynCommons, which tended to increase too much  
- licensed under a MPL/GPL/LGPL tri-license; version 1.18                                                                                                                                                                                                                     | 1721 |

FileServer class hierarchy

Objects implemented in the FileServer unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFileServer</td>
<td>A server to access SynFile data content</td>
<td>2536</td>
</tr>
</tbody>
</table>

TFileServer = class(TSQLRestserverDB)

A server to access SynFile data content

Server: TSQLHttpServer;

The running HTTP/1.1 server

constructor Create;

Create the database and HTTP/1.1 server

destructor Destroy; override;

Release used memory and data

function OnDatabaseUpdateEvent(Sender: TSQLRestServer; Event: TSQLEvent; aTable: TSQLRecordClass; const aID: TID; const aSentData: RawUTF8): boolean;

Database server-side trigger which will add an event to the TSQLAuditTrail table
procedure Event(Ctxt: TSQLRestServerURIContext);
   A RESTful service used from the client side to add an event to the TSQLAuditTrail table
   - an optional database record can be specified in order to be associated with the event

procedure AddAuditTrail(aEvent: TFileEvent; const aMessage: RawUTF8='';
aAssociatedRecord: TRecordReference=0);
   Add a row to the TSQLAuditTrail table
29.6. FileTables.pas unit

**Purpose:** SynFile ORM definitions shared by both client and server

### Units used in the FileTables unit

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1899</td>
</tr>
<tr>
<td>mORMoti18n</td>
<td>Internationalization (i18n) routines and classes for mORMot - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>2322</td>
</tr>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>717</td>
</tr>
<tr>
<td>SynCrypto</td>
<td>Fast cryptographic routines (hashing and cypher) - implements AES,XOR,ADLER32,MD5,RC4,SHA1,SHA256,SHA384,SHA512,SHA3 and JWT - optimized for speed (tuned assembler and SSE3/SSE4/AES-NI/PADLOCK support) - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1139</td>
</tr>
<tr>
<td>SynZip</td>
<td>Low-level access to ZLib compression (1.2.5 engine version) - this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18</td>
<td>1845</td>
</tr>
</tbody>
</table>

### Objects implemented in the FileTables unit

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFileRibbonTabParameters</td>
<td>The type of custom main User Interface description of SynFile</td>
<td>2539</td>
</tr>
<tr>
<td>TSQLAuditTrail</td>
<td>An AuditTrail table, used to track events and status</td>
<td>2539</td>
</tr>
<tr>
<td>TSQLData</td>
<td>An uncrypted Data table</td>
<td>2539</td>
</tr>
<tr>
<td>TSQLRecord</td>
<td>An abstract class, with common fields</td>
<td>2539</td>
</tr>
</tbody>
</table>

FileTables.pas unit - Rev. 1.18 Page 2539 of 2552
<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSQLMemo</td>
<td>An uncrypted Memo table</td>
<td>2539</td>
</tr>
<tr>
<td>TSQLSafeData</td>
<td>A crypted SafeData table</td>
<td>2539</td>
</tr>
<tr>
<td>TSQLSafeMemo</td>
<td>A crypted SafeMemo table</td>
<td>2539</td>
</tr>
</tbody>
</table>

**TSQLFile** = `class(TSQLRecordSigned)`

*An abstract class, with common fields*

**TSQLMemo** = `class(TSQLFile)`

*An uncrypted Memo table*
- will contain some text

**TSQLData** = `class(TSQLFile)`

*An uncrypted Data table*
- can contain any binary file content
- is also used as a parent for all cyphered tables (since the content is crypted, it should be binary, i.e. a BLOB field)

**TSQLSafeMemo** = `class(TSQLData)`

*A crypted SafeMemo table*
- will contain some text after AES-256 cypher
- just a direct sub class of TSQLData to create the "SafeMemo" table with the exact same fields as the "Data" table

**TSQLSafeData** = `class(TSQLData)`

*A crypted SafeData table*
- will contain some binary file content after AES-256 cypher
- just a direct sub class of TSQLData to create the "SafeData" table with the exact same fields as the "Data" table

**TSQLAuditTrail** = `class(TSQLRecord)`

*An AuditTrail table, used to track events and status*

**TFileRibbonTabParameters** = `object(TSQLRibbonTabParameters)`

*The type of custom main User Interface description of SynFile*

**Actions:** TFileActions;

*The SynFile actions*

**Types implemented in the FileTables unit**

**TFileAction** = ( faNoAction, faMark, faUnmarkAll, faQuery, faRefresh, faCreate, faEdit, faCopy, faExport, faImport, faDelete, faSign, faPrintPreview, faExtract, faSettings );

*The internal available actions, as used by the User Interface*

**TFileActions** = `set of` TFileAction;

*Set of available actions*

**TFileEvent** = ( feUnknownState, feServerStarted, feServerShutdown, feRecordCreated, feRecordDeleted, feRecordUpdated, feQueryModified, feQueryCompleted, feOperationCompleted, fePublicationPublished, feWizardStopped, feWizardStarted, feWizardClosed, feWizardOpened, feWizardProgress );

*The internal available events as used by the User Interface*
feRecordModified, feRecordDeleted, feRecordDigitallySigned, feRecordImported, feRecordExported);

*The internal events/states, as used by the TSQLAuditTrail table*

TPreviewAction = ( paPrint, paAsPdf, paAsText, paWithPicture, paDetails );

*Some actions to be used by the User Interface of a Preview window*

**Constants implemented in the FileTables unit**

DEF_ACTIONS = [faMark..faPrintPreview, faSettings];

*Some default actions, available for all tables*

DEF_ACTIONS_DATA = DEF_ACTIONS+[faExtract]-[faImport, faExport];

*Actions available for data tables (not for TSQLAuditTrail)*

DEF_SELECT = 'Name, Created, Modified, Keywords, SignatureTime';

*Default fields available for User Interface Grid*

FileActionsToolbar: array[0..3] of TFileActions = ( [faRefresh, faCreate, faEdit, faCopy, faExtract], [faExport..faPrintPreview], [faMark..faQuery], [faSettings] );

*Used to map which actions/buttons must be grouped in the toolbar*

FileActionsToolbar_MARKINDEX = 2;

FileActionsToolbar[FileActionsToolbar_MARKINDEX] will be the marked actions i.e. [faMark..faQuery]

FileTabs: array[0..4] of TFileRibbonTabParameters = ( (Table: TSQLAuditTrail; Select: 'Time, Status, StatusMessage'; Group: GROUP_MAIN; FieldWidth: 'gIZ'; ShowID: true; ReverseOrder: true; Layout: llClient; Actions: [faDelete, faMark, faUnmarkAll, faQuery, faRefresh, faPrintPreview, faSettings]), (Table: TSQLMemo; Select: DEF_SELECT; Group: GROUP_CLEAR; FieldWidth: 'IdId'; Actions: DEF_ACTIONS), (Table: TSQLData; Select: DEF_SELECT; Group: GROUP_CLEAR; FieldWidth: 'IdId'; Actions: DEF_ACTIONS_DATA), (Table: TSQLSafeMemo; Select: DEF_SELECT; Group: GROUP_SAFE; FieldWidth: 'IdId'; Actions: DEF_ACTIONS), (Table: TSQLSafeData; Select: DEF_SELECT; Group: GROUP_SAFE; FieldWidth: 'IdId'; Actions: DEF_ACTIONS_DATA) );

*This constant will define most of the User Interface property*

- the framework will create most User Interface content from the values stored within

GROUP_CLEAR = 1;

*Will define the 2nd User Interface ribbon group, i.e. uncrypted tables*

GROUP_MAIN = 0;

*Will define the first User Interface ribbon group, i.e. main tables*

GROUP_SAFE = 2;

*Will define the 3rd User Interface ribbon group, i.e. encrypted tables*

SERVER_HTTP_PORT = '888';

*The TCP/IP port used for the HTTP server*

- this is shared as constant by both client and server side
- in a production application, should be made customizable

**Functions or procedures implemented in the FileTables unit**
<table>
<thead>
<tr>
<th>Functions or procedures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateFileModel</td>
<td>Create the database model to be used</td>
<td>2541</td>
</tr>
</tbody>
</table>

```pascal
function CreateFileModel(Owner: TSQLRest): TSQLModel;

Create the database model to be used
- shared by both client and server sides
```
## 30. SWRS implications

**Software Architecture Design Reference Table**

The following table is a quick-reference guide to all the *Software Requirements Specifications* (SWRS) document items.

<table>
<thead>
<tr>
<th>SWRS #</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI-2.1.1</td>
<td>The framework shall be Client-Server oriented</td>
<td>2543</td>
</tr>
<tr>
<td>DI-2.1.1.1</td>
<td>A RESTful mechanism shall be implemented</td>
<td>2543</td>
</tr>
<tr>
<td>DI-2.1.1.2.1</td>
<td>Client-Server Direct communication shall be available inside the same process</td>
<td>2544</td>
</tr>
<tr>
<td>DI-2.1.1.2.2</td>
<td>Client-Server Named Pipe communication shall be made available by some dedicated classes</td>
<td>2544</td>
</tr>
<tr>
<td>DI-2.1.1.2.3</td>
<td>Client-Server Windows Messages communication shall be made available by some dedicated classes</td>
<td>2544</td>
</tr>
<tr>
<td>DI-2.1.1.2.4</td>
<td>Client-Server HTTP/1.1 over TCP/IP protocol communication shall be made available by some dedicated classes, and ready to be accessed from outside any <em>Delphi</em> Client (e.g. the implement should be AJAX ready)</td>
<td>2545</td>
</tr>
<tr>
<td>DI-2.1.2</td>
<td>UTF-8 JSON format shall be used to communicate</td>
<td>2545</td>
</tr>
<tr>
<td>DI-2.1.3</td>
<td>The framework shall use an innovative ORM (Object-relational mapping) approach, based on classes RTTI (Runtime Type Information)</td>
<td>2546</td>
</tr>
<tr>
<td>DI-2.1.4</td>
<td>The framework shall provide some Cross-Cutting components</td>
<td>2547</td>
</tr>
<tr>
<td>DI-2.1.5</td>
<td>The framework shall offer a complete SOA process</td>
<td>2547</td>
</tr>
<tr>
<td>DI-2.2.1</td>
<td>The <em>SQLite3</em> engine shall be embedded to the framework</td>
<td>2548</td>
</tr>
<tr>
<td>DI-2.2.2</td>
<td>The framework libraries, including all its <em>SQLite3</em> related features, shall be tested using Unitary testing</td>
<td>2549</td>
</tr>
<tr>
<td>DI-2.3.1.1</td>
<td>A Database Grid shall be made available to provide data browsing in the Client Application - it shall handle easy browsing, by column resizing and sorting, on the fly customization of the cell content</td>
<td>2549</td>
</tr>
</tbody>
</table>
30.1. Client Server ORM/SOA framework

30.1.1. SWRS # DI-2.1.1

The framework shall be Client-Server oriented

*Design Input 2.1.1 (Initial release): The framework shall be Client-Server oriented.*

Client–Server model of computing is a distributed application structure that partitions tasks or workloads between service providers, called servers, and service requesters, called clients.

Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server machine is a host that is running one or more server programs which share its resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await (listen for) incoming requests.

The *Synopses mORMot Framework* shall implement such a Client-Server model by a set of dedicated classes, over various communication protocols, but in an unified way. Application shall easily change the protocol used, just by adjusting the class type used in the client code. By design, the only requirement is that protocols and associated parameters are expected to match between the Client and the Server.

*This specification is implemented by the following units:*  

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>
| mORMot   | Common ORM and SOA classes for mORMot  
*See in particular TSQLRecord, TSQLRest, TSQLRestServer, TSQLRestClientURI and TSQLTableJSON.* | 1899 |

30.1.2. SWRS # DI-2.1.1.1

A RESTful mechanism shall be implemented

*Design Input 2.1.1.1 (Initial release): A RESTful mechanism shall be implemented.*

REST-style architectures consist of clients and servers, as was stated in SWRS # DI-2.1.1. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of "representations" of "resources". A resource can be
essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource.

In the *Synopsen mORMot Framework*, so called "resources" are individual records of the underlying database, or list of individual fields values extracted from these databases, by a SQL-like query statement.

**This specification is implemented by the following units:**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>See in particular TSQLRest, TSQLRestServer and TSQLRestClientURI.</td>
<td></td>
</tr>
</tbody>
</table>

30.1.3. SWRS # DI-2.1.1.2.1

**Client-Server Direct communication shall be available inside the same process**

*Design Input 2.1.1.2 (Initial release): Communication should be available directly in the same process memory, or remotely using Named Pipes, Windows messages or HTTP/1.1 protocols.*

Client-Server Direct communication shall be available inside the same process.

**This specification is implemented by the following units:**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>See in particular TSQLRestServer, URIRequest, USEFASTM4ALLOC and TSQLRestClientURIDll.Create.</td>
<td></td>
</tr>
</tbody>
</table>

30.1.4. SWRS # DI-2.1.1.2.2

**Client-Server Named Pipe communication shall be made available by some dedicated classes**

Client-Server Named Pipe communication shall be made available by some dedicated classes.

**This specification is implemented by the following units:**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>See in particular TSQLRestServer.ExportServerNamedPipe, TSQLRestClientURINamedPipe.Create and TSQLRestClientURI.</td>
<td></td>
</tr>
</tbody>
</table>

30.1.5. SWRS # DI-2.1.1.2.3

**Client-Server Windows Messages communication shall be made available by some dedicated classes**
Client-Server Windows Messages communication shall be made available by some dedicated classes.

This specification is implemented by the following units:

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>See in particular TSQLRestClientURI, TSQLRestServer.ExportServerMessage and TSQLRestClientURIMessage.Create.</td>
<td></td>
</tr>
</tbody>
</table>

30.1.6. SWRS # DI-2.1.1.2.4

Client-Server HTTP/1.1 over TCP/IP protocol communication shall be made available by some dedicated classes, and ready to be accessed from outside any Delphi Client (e.g. the implement should be AJAX ready)

Client-Server HTTP/1.1 over TCP/IP protocol communication shall be made available by some dedicated classes, and ready to be accessed from outside any Delphi Client (e.g. the implement should be AJAX ready).

This specification is implemented by the following units:

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>See in particular TSQLRestClientURI and TSQLRestServer.URI.</td>
<td></td>
</tr>
<tr>
<td>mORMotHttpServer</td>
<td>HTTP/1.1 RESTFUL JSON Server classes for mORMot</td>
<td>2313</td>
</tr>
<tr>
<td></td>
<td>See in particular TSQLHttpServer.Create, TSQLHttpServer.DBServer and TSQLHttpServer.AddServer.</td>
<td></td>
</tr>
<tr>
<td>SynCrtSock</td>
<td>Classes implementing TCP/UDP/HTTP client and server protocol</td>
<td>1083</td>
</tr>
<tr>
<td>mORMotHttpClient</td>
<td>HTTP/1.1 RESTful JSON Client classes for mORMot</td>
<td>2306</td>
</tr>
<tr>
<td></td>
<td>See in particular TSQLHttpClient.Create.</td>
<td></td>
</tr>
</tbody>
</table>

30.1.7. SWRS # DI-2.1.2

UTF-8 JSON format shall be used to communicate

Design Input 2.1.2 (Initial release): UTF-8 JSON format shall be used to communicate.

JSON, as defined in the Software Architecture Design (SAD) document, is used in the Synopse mORMot Framework for all Client-Server communication. JSON (an acronym for JavaScript Object Notation) is a lightweight text-based open standard designed for human-readable data interchange. Despite its relationship to JavaScript, it is language-independent, with parsers available for virtually every programming language.
JSON shall be used in the framework for returning individual database record content, in a disposition which could make it compatible with direct JavaScript interpretation (i.e. easily creating JavaScript object from JSON content, in order to facilitate AJAX application development). From the Client to the Server, record content is also JSON-encoded, in order to be easily interpreted by the Server, which will convert the supplied field values into proper SQL content, ready to be inserted to the underlying database.

JSON should be used also within the transmission of request rows of data. It therefore provide an easy way of data forming between the Client and the Server.

The Synopse mORMot Framework shall use UTF-8 encoding for the character transmission inside its JSON content. UTF-8 (8-bit Unicode Transformation Format) is a variable-length character encoding for Unicode. UTF-8 encodes each character (code point) in 1 to 4 octets (8-bit bytes). The first 128 characters of the Unicode character set (which correspond directly to the ASCII) use a single octet with the same binary value as in ASCII. Therefore, UTF-8 can encode any Unicode character, avoiding the need to figure out and set a "code page" or otherwise indicate what character set is in use, and allowing output in multiple languages at the same time. For many languages there has been more than one single-byte encoding in usage, so even knowing the language was insufficient information to display it correctly.

This specification is implemented by the following units:

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>See in particular TTextWriter.Create, TTextWriter.AddJSONEscape, IsJSONString, JSONDecode, JSONEncode, JSONEncodeArray, GetJSONField and JSON_CONTENT_TYPE.</td>
</tr>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>See in particular TSQLTable.GetJSONValues, TSQLTableJSON.Create, TSQLTableJSON.UpdateFrom, TJSONWriter.Create, TSQLRecord.CreateJSONWriter, TSQLRecord.GetJSONValues, GetJSONObjectAsSQL and UnJSONFirstField.</td>
</tr>
</tbody>
</table>

30.1.8. SWRS # DI-2.1.3

The framework shall use an innovative ORM (Object-relational mapping) approach, based on classes RTTI (Runtime Type Information)

Design Input 2.1.3 (Initial release): The framework shall use an innovative ORM (Object-relational mapping) approach, based on classes RTTI (Runtime Type Information).

ORM, as defined in the Software Architecture Design (SAD) document, is used in the Synopse mORMot Framework for accessing data record fields directly from Delphi Code.

Object-relational mapping (ORM, O/RM, and O/R mapping) is a programming technique for converting data between incompatible type systems in relational databases and object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the Delphi programming language.
The published properties of classes inheriting from a new generic type named TSQLRecord are used to define the field properties of the data. Accessing database records (for reading or update) shall be made by using these classes properties, and some dedicated Client-side methods.

**This specification is implemented by the following units:**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMot</td>
<td>Common ORM and SOA classes for mORMot</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td><em>See in particular</em> TClassProp, TClassType, TEnumType, TTypeInfo, TSQLRecord.ClassProp, TSQLRecord.GetJSONValues, TPropInfo.GetValue, TPropInfo.SetValue and TSQLRecordProperties.</td>
<td></td>
</tr>
</tbody>
</table>

### 30.1.9. SWRS # DI-2.1.4

**The framework shall provide some Cross-Cutting components**

*Design Input 2.1.4 (Initial release): The framework shall provide some Cross-Cutting components.*

Cross-Cutting infrastructure layers shall be made available for handling data filtering and validation, security, session, cache, logging and testing (framework uses test-driven approach and features stubbing and mocking).

All crosscutting scenarios are coupled, so you benefit of consisting APIs and documentation, a lot of code-reuse, JSON/RESTful orientation from the ground up.

**This specification is implemented by the following units:**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynCommons</td>
<td>Common functions used by most Synopse projects</td>
<td>717</td>
</tr>
<tr>
<td></td>
<td><em>See in particular</em> TSynLog and PatchCodePtrUInt.</td>
<td></td>
</tr>
</tbody>
</table>

### 30.1.10. SWRS # DI-2.1.5

**The framework shall offer a complete SOA process**

*Design Input 2.1.5 (Initial release): The framework shall offer a complete SOA process.*

In order to follow a Service Oriented Architecture design, your application's business logic can be implemented in several ways using mORMot:

- Via some TSQLRecord inherited classes, inserted into the database model, and accessible via some RESTful URI - this is implemented by our ORM architecture - see SWRS # DI-2.1.3;
- By some RESTful services, implemented in the Server as published methods, and consumed in the Client via native Delphi methods;
- Defining some RESTful service contracts as standard Delphi interface, and then run it seamlessly on both client and client sides.

**This specification is implemented by the following units:**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>

SAD - SWRS implications - Rev. 1.18  Page 2548 of 2552
30.2. SQLite3 engine

30.2.1. SWRS # DI-2.2.1
The SQLite3 engine shall be embedded to the framework

*Design Input 2.2.1 (Initial release): The SQLite3 engine shall be embedded to the framework.*

The SQLite3 database engine is used in the Synapse mORMot Framework as its kernel database engine. SQLite3 is an ACID-compliant embedded relational database management system contained in a C programming library.

This library shall be linked statically to the Synapse mORMot Framework, or using official external sqlite3.dll distribution, and interact directly from the Delphi application process.

The Synapse mORMot Framework shall enhance the standard SQLite3 database engine by introducing some new features stated in the Software Architecture Design (SAD) document, related to the Client-Server purpose or the framework - see SWRS # DI-2.1.1.

**This specification is implemented by the following units:**

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mORMot</strong></td>
<td>Common ORM and SOA classes for mORMot See in particular TServiceFactory, TServiceFactoryServer, TServiceFactoryClient, TServiceContainerClient, TServiceContainerServer and TServiceContainer.</td>
<td>1899</td>
</tr>
<tr>
<td><strong>SynSQLite3</strong></td>
<td>SQLite3 Database engine direct access See in particular TSQLite3LibraryDynamic, TSQLite3Library, TSQLRequest.Execute, TSQLDataBase, TSQLTableDB.Create, TSQLite3Blob, TSQLite3DB, TSQLite3FunctionContext, TSQLite3Statement, TSQLite3Value, TSQLite3ValueArray, TSQLTableDB, TSQLRequest, TSQLBlobStream and ESQLException.</td>
<td>1646</td>
</tr>
<tr>
<td><strong>SynSQLite3Static</strong></td>
<td>SQLite3 3.33.0 Database engine - statically linked for Windows/Linux See in particular TSQLite3LibraryStatic.</td>
<td>1711</td>
</tr>
<tr>
<td><strong>mORMotSQLite3</strong></td>
<td>SQLite3 embedded Database engine used as the mORMot SQL kernel See in particular TSQLRestServerDB and TSQLRestClientDB.</td>
<td>2382</td>
</tr>
<tr>
<td><strong>SynDBSQLite3</strong></td>
<td>SQLite3 direct access classes to be used with our SynDB architecture See in particular TSQLDBSQLite3Connection, TSQLDBSQLite3Statement and TSQLDBSQLite3ConnectionProperties.</td>
<td>1298</td>
</tr>
</tbody>
</table>
30.2.2. SWRS # DI-2.2.2

The framework libraries, including all its SQLite3 related features, shall be tested using Unitary testing

*Design Input 2.2.2 (Initial release)*: The framework libraries, including all its SQLite3 related features, shall be tested using Unitary testing.

The Synopses mORMot Framework shall use all integrated Unitary testing features provided by a common testing framework integrated to all Synopses products. This testing shall be defined by classes, in which individual published methods define the actual testing of most framework features.

All testing shall be run at once, for example before any software release, or after any modification to the framework code, in order to avoid most regression bugs.

This specification is implemented by the following units:

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynSelfTests</td>
<td>Automated tests for common units of the Synopses mORMot Framework</td>
<td>1528</td>
</tr>
<tr>
<td></td>
<td>See in particular TTestLowLevelCommon, TTestLowLevelTypes,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TTestBasicClasses, TTestSQLite3Engine, TTestFileBased,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TTestMemoryBased, TTestFileBasedWAL and TTestClientServerAccess.</td>
<td></td>
</tr>
</tbody>
</table>

30.3. User interface

30.3.1. SWRS # DI-2.3.1.1

A Database Grid shall be made available to provide data browsing in the Client Application - it shall handle easy browsing, by column resizing and sorting, on the fly customization of the cell content

*Design Input 2.3.1 (Initial release)*: An User Interface, with buttons and toolbars shall be easily being created from the code, with no RAD needed, using RTTI and data auto-description.

A Database Grid shall be made available to provide data browsing in the Client Application - it shall handle easy browsing, by column resizing and sorting, on the fly customization of the cell content.

This specification is implemented by the following units:

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMotUI</td>
<td>Grid to display database content for mORMot</td>
<td>2404</td>
</tr>
<tr>
<td></td>
<td>See in particular TSQLTableToGrid.Create.</td>
<td></td>
</tr>
<tr>
<td>mORMotToolBar</td>
<td>ORM-driven Office 2007 Toolbar for mORMot</td>
<td>2390</td>
</tr>
<tr>
<td></td>
<td>See in particular TSQLLister.Create.</td>
<td></td>
</tr>
</tbody>
</table>

30.3.2. SWRS # DI-2.3.1.2

Toolbars shall be able to be created from code, using RTTI and enumerations types
for defining the action

Toolbars shall be able to be created from code, using RTTI and enumerations types for defining the action.

This specification is implemented by the following units:

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMoToolBar</td>
<td>ORM-driven Office 2007 Toolbar for mORMot</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See in particular TSQLRibbon.Create, TSQLRibbonTab, TSQLLister and TSQLCustomToolBar.Init.</td>
<td>2390</td>
</tr>
</tbody>
</table>

30.3.3. SWRS # DI-2.3.1.3

Internationalization (i18n) of the whole User Interface shall be made available by defined some external text files: Delphi resourcestring shall be translatable on the fly, custom window dialogs automatically translated before their display, and User Interface generated from RTTI should be included in this i18n mechanism.

Internationalization (i18n) of the whole User Interface shall be made available by defined some external text files: Delphi resourcestring shall be translatable on the fly, custom window dialogs automatically translated before their display, and User Interface generated from RTTI should be included in this i18n mechanism.

This specification is implemented by the following units:

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMoti18n</td>
<td>Internationalization (i18n) routines and classes for mORMot</td>
<td>2322</td>
</tr>
<tr>
<td></td>
<td>See in particular TLanguage, TLanguageFile.Create, S2U, U2S,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TLanguageFile.StringToUTF8, TLanguageFile.TimeToText,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TLanguageFile.DateToText,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TLanguageFile.DateTimeToText,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TLanguageFile.UTF8ToString,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TLanguageFile.Translate and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

30.3.4. SWRS # DI-2.3.2

A reporting feature, with full preview and export as PDF or TXT files, shall be integrated

Design Input 2.3.2 (Initial release): A reporting feature, with full preview and export as PDF or TXT files, shall be integrated.

The Synopse mORMot Framework shall provide a reporting feature, which could be used stand-alone, or linked to its database mechanism. Reports shall not be created using a RAD approach (e.g. defining bands and fields with the mouse on the IDE), but shall be defined from code, by using some dedicated methods, adding text, tables or pictures to the report. Therefore, any kind of report shall be generated.

This reports shall be previewed on screen, and exported as PDF or TXT on request.
This specification is implemented by the following units:

<table>
<thead>
<tr>
<th>Unit Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mORMotReport</td>
<td>Reporting unit</td>
<td>2352</td>
</tr>
<tr>
<td></td>
<td><em>See in particular</em> TGDIPages.</td>
<td></td>
</tr>
<tr>
<td>SynGdiPlus</td>
<td>GDI+ library API access</td>
<td>1350</td>
</tr>
<tr>
<td></td>
<td><em>See in particular</em> TGDIPPlus.DrawAntiAliased.</td>
<td></td>
</tr>
<tr>
<td>SynPdf</td>
<td>PDF file generation</td>
<td>1473</td>
</tr>
<tr>
<td></td>
<td><em>See in particular</em> TPdfDocument and TPdfCanvas.RenderMetaFile.</td>
<td></td>
</tr>
</tbody>
</table>